THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Mesoscopic Thin Film Superconductors

A Computational Framework

Mikael Håkansson

Applied Quantum Physics Laboratory
Department of Microtechnology and Nanoscience - MC2
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2015

Mesoscopic Thin Film Superconductors - A Computational Framework
MIKAEL HÅKANSSON

Applied Quantum Physics Laboratory
Department of Microtechnology and Nanoscience - MC2
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31 772 1000
www.chalmers.se

Author email: mikael.hakansson@gmail.com

ISSN 1652-0769
Technical Report MC2-300

Cover: Spectral plot of local density of states along the edge of the d-wave superconductor in Figure 5.10.

Printed by Chalmers Reproservice
Göteborg, Sweden 2015

Mesoscopic Thin Film Superconductors - A Computational Framework

MIKAEL HÅKANSSON

Applied Quantum Physics Laboratory
Department of Microtechnology and Nanoscience - MC2
Chalmers University of Technology

Abstract

Motivated by experiments on superconductors in the microscopic regime and the realization of small superconducting devices such as qubits, we initiate a computational project in the form of a numerical simulation package to model and predict the behavior of these type of systems. It is a multidisciplinary endeavor in that it employs theory, computational science, software design, and to some extent new visualization techniques. Our intention is to create a flexible and modular code library which is capable of simulating a wide variety of superconducting systems, while significantly shortening the start-up time for computational projects.

We base the numerical model in quasiclassical theory with the Eilenberger transport equation, and execute the highly parallel computations on a contemporary graphics card (GPU). In this thesis we touch on the theoretical background, describe the discretization of the theory, and present a thorough overview on how to solve the equations in practice for a general 2-dimensional geometry, as well as review the design principles behind the developed software. Moreover, a few selected results are presented to show that the output is sound, and to highlight some new findings. In paper I we examine a new low temperature phase in which spontaneous edge currents emerge in a d-wave superconducting square grain, breaking time-reversal symmetry. The modeling of such a system is made possible by our development of a simulation domain description which allows for specular boundary conditions in an arbitrarily shaped geometry.

Keywords: Superconductivity, quasiclassical theory, Eilenberger equation, Riccati equation, time-reversal symmetry, coherence functions, computational, simulation, CUDA, GPU

Acknowledgements

First of all I would like to thank Mikael Fogelström and Tomas Löfwander for taking me on and letting me work in the fascinating field of superconductivity. I really appreciate all the time you spent on our discussions, where the topic oscillated between physics and programming in a chaotic manner, as it should do. I'm also thankful for the great responsibility you trusted me with, letting me rewrite the entire code for the nth time.

It has also been a true pleasure working with and alongside all the great people that make our group. It has given me many good moments with inspiring discussions on just about any topic, plenty of good laughs, cake, many lost table tennis games, the finest Argentinian cuisine, more cake, an epic ski journey, and the unrelenting quest in how to acquire the best coffee the office has to offer.

Thank you!

Göteborg, January 2015
Mikael Håkansson

List of publications

This thesis is based mainly on the work behind developing a simulation framework, which in turn was used in uncovering the physics investigated in the following paper:

I. **Spontaneously broken time-reversal symmetry in high-temperature superconductors**
Mikael Håkansson, Tomas Löfwander, and Mikael Fogelström
Under review (arXiv:1411.0886 [cond-mat.supr-con])

Contents

List of figures

Nomenclature

Abbreviations

[G,T]FLOPS	[Giga,Tera]Floating point operations per second
API	Application programming interface
CPU	Central processing unit
DOS	Density of states
GB	Gigabyte
GPU	Graphics processing unit
LDOS	Local density of states
MB	Megabyte
OOP	Object oriented programming
STL	Standard template library
YBCO	Yttrium barium copper oxide

(i,j)	Discrete lattice coordinate
Δ	Superconducting order parameter
Δ'	Order parameter in a rotational/local frame
$\Delta(i,j)$	Order parameter value at discrete lattice coordinate (i,j)
$\delta\Omega$	Deviation from normal-state free energy
ϵ	Energy
γ	Coherence function for particle
$\gamma(i,j)$	Coherence function value at discrete lattice coordinate (i,j)
γ_h	Bulk solution for γ
Γ	Specularly reflected (from boundary) trajectory, exact
$\tilde{\gamma}$	Coherence function for hole
$\tilde{\gamma}_h$	Bulk solution for $\tilde{\gamma}$
$\hat{1}$	Identity matrix in spin space
$\hat{\boldsymbol{p}}_F$	Normalized Fermi momentum
$\hat{\boldsymbol{v}}_F$	Normalized Fermi velocity
$\hat{\tau}$	Pauli matrix in spin space
$\hat{g}$	Quasiclassical matrix propagator / Green's functions

$\hbar$	Reduced Planck constant
$\boldsymbol{A}$	Magnetic vector potential
$\boldsymbol{j}$	Quasiparticle current
$\boldsymbol{k}$	Wave-vector
$\boldsymbol{p}_F$	Fermi momentum
$\boldsymbol{p}_s$	Superfluid momentum
$\boldsymbol{R}$	Position vector
$\boldsymbol{v}_F$	Fermi velocity
λ	Coupling constant
$\mathcal{Y}$	Superconducting symmetry basis function
Φ_0	Flux quantum
Θ_i	Angles of the Fermi velocity from the discrete set
θ_p	Angle between Fermi momentum and x-axis
θ_{v_F}	Angle between Fermi velocity and x-axis
ε_n	Matsubara frequency
ξ_0	Coherence length
c	Speed of light
e	Electron charge
f	Anomalous Green's function
g	Green's function
g_θ	Green's function for momentum direction θ, in reference frame
g'_θ	Green's function for momentum direction θ, in a rotational/local frame
H	Magnetic field
h	Planck constant
H_c	Critical magnetic field
h_i	Physical distance between the last interior lattice point and the actual geometry boundary (along the Riccati trajectory)
h_o	Physical distance between actual geometry boundary and the first interior lattice point (along the Riccati trajectory)
H_{c1}	Lower critical magnetic field
H_{c2}	Upper critical magnetic field
k_B	Boltzmann constant
N	DOS
N_F	Normal-state DOS at the Fermi level
N_x	Number of discrete lattice units in x
N_y	Number of discrete lattice units in y
N_ϵ	Number of Matsubara or real energies
T	Temperature
T_c	Critical temperature

Chapter 1

Introduction

In April 1911, Heike Kamerlingh Onnes conducted an experiment where he measured the temperature dependence of electrical resistance in mercury [1]. Unexpectedly, he found that below a certain temperature the resistance vanished abruptly and completely. This newly discovered resistance-free state, entered through a second order phase transition, was shortly thereafter dubbed superconductivity. The material used in this seminal experiment was mercury, and the temperature was a relatively cool 4.2K. It was soon discovered that many other metals exhibit the same transition into a superconducting state, albeit at slightly different temperatures. The transition temperature, also known as the critical temperature, was, however, always found to be below 20K. Being able to carry an electrical current with zero resistance is not the only hallmark of a superconductor. What sets it apart from a perfect conductor is the expulsion of magnetic field from its interior, a property known as the Meissner effect, and manifests only in superconducting materials below the critical temperature [2].

The origin of superconductivity was unknown until Bardeen, Cooper and Schrieffer introduced their complete microscopic theory of superconductivity, also known as BCS theory, in 1957 [3]. In 1972 they received a Nobel prize for their work. BCS theory predicted a maximum critical temperature of about 30K, which corresponded well with observations to date. However, in 1986 Bednorz and Müller from IBM found a ceramic material with a critical temperature of 35K [4], a discovery which awarded them the Nobel prize the following year. Not long after, it was found that a modification to the ceramic material raised the critical temperature significantly higher to 92K [5]. In this respect, these new materials clearly did not obey the prediction of BCS theory. The latter material is an yttrium based cuprate perovskite, more commonly known as YBCO, and is today one of the most widely used superconducting materials. A range of other material types exhibiting superconductivity has since been discovered, with physical signatures indicating yet different modes of superconductivity. The types of superconductors first discovered, i.e. elemental metals which can be explained by

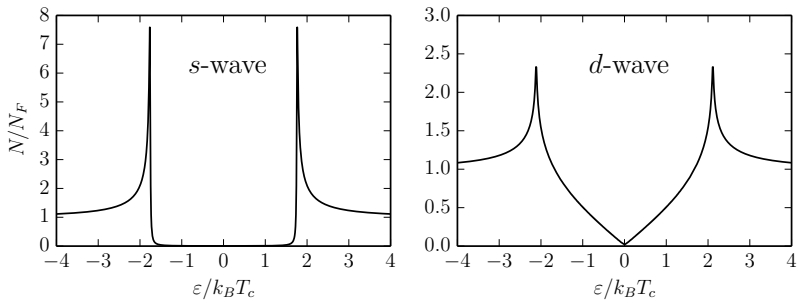

Figure 1.1: *Example of a density of states spectrum for a conventional superconductor (left), and an unconventional superconductor (right).*

BCS theory, are called *conventional superconductors*, while the rest fall into the category of *unconventional superconductors*. As of this date, a complete explanation of the microscopic interaction(s) among the unconventional superconductors is yet to be found.

Central to the BCS model is the notion of electrons pairing up two and two into so called *Cooper pairs*. The pairing occurs for electrons possessing opposite spin and momentum, effectively turning two fermions into a boson-like state. The connective force behind the pairing is said to be mediated by phonons. More illustratively, as the electrons move through the lattice they interact with the ions and leave a wake of charge perturbations. Pairs of electrons can then couple to each others wakes by Coulomb interaction, in such a way as to enable them to move unhindered through the lattice. From an energy point of view, the pairing, occurring between two electrons near the Fermi surface, brings a lowering of the free energy. As such the Fermi sea becomes unstable below the critical temperature, and an energy gap opens up since previously excitable electrons are now bound into Cooper pairs. To excite electrons into the normal conducting state, there is now an associated energy threshold. Since no single electrons can be excited below this energy, the density of states (DOS) is zero in this range. This is known as the superconducting energy gap. The states previously available in the gap are pushed to just outside the gap, producing a characteristic measurable density of states profile (Figure 1.1).

Depending on the material of the superconductor, the mechanism and behavior of the electron pairing differs. The phonon mediated pairing described above is a property of the first discovered low temperature elemental superconductors, and is known as s-wave pairing symmetry. Here the Cooper pairs are free to move in any direction, and as such the pairing is isotropic and the superconducting order parameter is not dependent on the wave vector $\boldsymbol{k}$. The high temperature YBCO superconductors, however, are believed to have a predominantly anisotropic d-

wave symmetry, where pairing is only allowed along the crystal axes [6]. Among the most compelling evidence is the so-called tricrystal experiment [7], which forces any d-wave superconductor into a unique state which is unattainable with conventional s-wave pairing. The precise nature of the mediating force is not fully known, although there are a few candidate theories, e.g. spin density waves [8]. This naturally gives rise to a $\boldsymbol{k}$-dependent order parameter, with nodes where the $\boldsymbol{k}$-vector is maximally misaligned with the crystal axes. The DOS profile in and around the energy gap is unique to each pairing symmetry, and is an important signature in identifying the pairing symmetry for a superconductor. Figure 1.1 (right) displays the DOS spectrum for a d-wave superconductor.

The total spin of a Cooper pair in s- and d-wave pairing is $S = 0$, i.e. the singlet state. There are other superconductors with electron pairing such that the total spin equals $S = 1$ (spin triplet). These have p-wave pairing symmetry, but are not covered in this thesis.

As mentioned earlier, an applied magnetic field will be expelled from the interior if the material is below the critical temperature T_c. However, this is only true up to a certain field strength H_c, known as the critical field. The response of a superconductor exposed to a field exceeding this depends on the material. For one group of materials, typically elemental metals, the superconductivity quickly breaks down above H_c. These are called Type-I superconductors. For most of the unconventional superconductors, two critical fields strengths $H_{c1} < H_{c2}$ are attributed. Below H_{c1}, the behavior is like Type-I, i.e. we have the *Meissner state* of complete expulsion. When exceeding H_{c1}, however, magnetic flux starts entering the material, as carried by supercurrent vortices. Each vortex carries a fixed quantum of flux, and goes by the name of flux vortices, or Abrikosov vortices [9]. These can be considered quasiparticles with long range attraction and short range repulsion, and form a triangular lattice in a clean material. At the center of each vortex the superconductivity breaks down, allowing for normal states to be exited [10], or for other modes of superconductivity to emerge [11, 12]. When the material is carrying these vortices, it is said to be in the *mixed state*. As the magnetic field increases, more vortices enter the superconductor, up until the second critical field is reached when the superconductivity breaks down. This is known as Type-II behavior. The different behavior between the two hinges upon the free energy description. In Type-I superconductors, the free energy is minimized by minimizing the amount of interface between the normal and superconducting (NS) phases. In contrast, for Type-II superconductors the free energy is minimized by maximizing the NS interface area, as manifested by the formation of Abrikosov vortices [13].

Superconductivity is very much an area of active research, as there are still many unanswered questions such as the exact nature of the pairing symmetry in cuprates. There is also mounting evidence of a low-temperature time-reversal symmetry breaking phase in certain types of superconductors [14–19]. On the

other hand, the corresponding magnetic field such a phase would give rise to has not been observed in experiments [20–24]. Experiments are being conducted on nano-scale superconductors both from a basic research standpoint, but also more specifically as a means of realizing qubits for quantum computing and other superconducting devices. On this small spatial scale measurement is difficult, and it is not always possible to measure all the quantities desired to interpret the results that indeed was acquired. In light of this, we were motivated to develop a complementary research tool in the form of a computational software package for simulating mesoscopic thin film superconducting grains. This could be used entirely on its own to investigate interesting regimes, but also to run simulations alongside experiment and exchange data to hopefully get a deeper understanding of the physical picture.

There are various theories of superconductivity, most notably the phenomenological Ginzburg-Landau theory, BCS microscopic theory, and quasiclassical theory, with each being suitable to different regimes. Considering research and experiment on the nano and micrometer scale, we chose to apply the quasiclassical picture. Here we abandon the atomistic effects and quantum fluctuations, but retain enough physics to describe a rich picture on the intended scale. On this length scale of the *coherence length*, $\xi_0 \sim \hbar v_F/\Delta$, interference effects due to confinement are possible, giving rise to interesting phenomena which we want to capture. This regime is well described in the quasiclassical picture, and is computationally feasible. With a microscopic theory, however, computational demands may be too high to model systems of this size, given our resources.

1.1 Software design goals

When designing a code library for a particular implementation, computational or other, there are naturally a multitude of different aspects to consider. Of central importance is to decide the degree of generalization for the problem at hand. To solve a specific problem with known input parameters, there is usually little or no need for generalization, and the code development will be relatively swift. However, the drawback with this approach becomes apparent if one would like to solve a slightly different but related problem. In this case a potentially large rework of the code base might be required. We will also end up with multiple copies of similar code, which can be cumbersome to maintain. At the other end of the spectrum we can anticipate and allow for all sort of variations of the problem. This is of course very desirable as an end product, but the development cost will be high since the coding complexity increases with generalization, and in the end we might not even end up needing all the functionality implemented. Naturally, the best strategy often lies somewhere between these two extremes. Significant deciding factors could be e.g. how much time can be spent on development, estimated time to have some desired results, a known subclass of problems which are of particular interest and should be prioritized.

The first goal set for this project was to not go down the path of specialization. Although first results would probably arrive relatively fast, this can easily result in an ad hoc code base which can not be expanded upon. It will also be difficult later on for other researchers to take over or utilize the work already made. Instead, we decided to develop a more flexible long term solution and create a code library structured in such a way that code already written could be reused, and new features could be added at any point in time. Moreover, another important property we decided on is that the low level code, the implementation details, should be hidden from the user. The "language" of the resulting code should closely resemble that of the physics involved, and one should not need to delve into arcane coding syntax to set up a simulation. As such, some of the key concepts for our software design are modularity, expandability, and usability, which of course are desirable generic properties for any long term software project.

Previous computational work applying quasiclassical theory of superconductivity are plenty and illuminating, for a few examples see e.g. [11, 12, 25, 26]. Thanks to the collaborative efforts of the research community, robust numerical methods have been developed. But to the best of our knowledge, an approach toward a flexible tool as described above has not been done yet, and we believe there is great potential in creating one. Moreover, while there has been work done on involving finite system boundary conditions, it has been implemented for special cases such as translationally invariant systems with a straight boundary [25], or for a fixed shape like a disc [26]. Our implementation will allow for a finite grain (vacuum-superconducting interface) with a completely general boundary shape, where the simulation domain can be either simply connected or multiply connected.

1.2 Overview of thesis

With some background and motivation covered, this chapter is concluded by giving an overview of the rest of the thesis.

In chapter 2 we present a few fundamental equations and relations of quasiclassical theory, with a focus on the steps that takes us from the governing equation all the way to the numerical expressions as they are used in the implementation. The reason for this perhaps somewhat pragmatic approach to the theoretical background is that we would like to spend more effort on reporting on the main work behind this thesis; the computational aspects like software design features and implementation details, as well as the results acquired from the use of the developed software. This way, the reader will be not only introduced to the physical backdrop of the appended paper, but also informed on what other types of problems can be solved by using our tool.

Motivation as to why we chose the programming languages we did for the implementation is put forward in chapter 3, while also explaining some key concepts needed to understand the API structure.

This leads us to the 4th chapter, which covers the implementation specifics. A detailed description is given on how we use the results in chapter 2 in practice, in order to build the foundation of our numerical solver. Moreover, the main building blocks of our software package are outlined, with the purpose of introducing the reader to the software design and attempt to instill some idea on what is possible to do with it today, and also to what extent it can be expanded from its current form. Finally, a thoroughly commented code example is listed where a superconducting system (grain), identical to the one investigated in paper I, is set up from scratch in order to expose the naked truth on how to use the software.

Chapter 5 presents some of the results acquired by the use of our code library. Results are given first in terms of validation signatures, like a DOS spectrum, to prove that it reproduces known results and to introduce physical concepts discussed in paper I. Moreover, the main results in paper I is also briefly discussed, as an introduction to the paper itself.

Finally, in chapter 6 we summarize the key results of this thesis work, and provide examples of some new features to implement which would allow numerical studies in new interesting physical regimes.

Chapter 2

The Eilenberger Equation

In this thesis we use quasiclassical theory of superconductivity [27–29]. Central to this theory is the Eilenberger transport equation and the Green's functions. There is plenty of material on this topic, as such the full description and derivation will not be repeated here. The interested reader is instead referred to the references above for a comprehensive background. For this work, the modern formulation by Eschrig [30] has been used. Following is a brief summary of some important concepts and equations necessary to follow the numerical treatment.

The Eilenberger equations reads

$$[\epsilon\hat{\tau}_3 - \hat{\Delta},\hat{g}] + i\hbar\boldsymbol{v}_F \cdot \nabla\hat{g} = 0, \tag{2.1}$$

with the normalization condition $\hat{g}^2 = -\pi^2\hat{1}$, where $\hat{g}$ is the quasiclassical Green's function (also known as the quasiclassical matrix propagator), and $\hat{\Delta}$ is the particle-hole self energy

$$\hat{g} = \begin{pmatrix} g & f \\ \tilde{f} & -g \end{pmatrix}, \quad \hat{\Delta} = \begin{pmatrix} 0 & \Delta \\ \Delta^\dagger & 0 \end{pmatrix}, \tag{2.2}$$

and $\hat{\tau}_3$ is the Pauli matrix in spin space. From the elements of $\hat{g}$ we can compute many physical properties of the particular system we are interested in. The general form of many of these properties include an integration over the Fermi surface, and a summation over the Matsubara frequencies $i\varepsilon_n$. For properties where the spectral distribution is of interest, real energies $\epsilon = \varepsilon + i\delta^+$ are used with no summation. Below, expressions are listed for a few physical properties.

The order parameter can be computed by rearranging the superconducting gap equation on the following form

$$\Delta(\boldsymbol{R}) = \lambda T \sum_{\varepsilon_n < \varepsilon_c} \langle \mathcal{Y}^*(\theta_{\hat{\boldsymbol{p}}_F}) f(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\varepsilon_n)\rangle_{\hat{\boldsymbol{p}}_F}, \tag{2.3}$$

where $\langle\rangle_{\hat{\boldsymbol{p}}_F} = \int \frac{d\theta_p}{2\pi}$ denotes the average of the momentum direction on the Fermi surface, where θ_p is the angle between the momentum direction and the crystal a-axis, and

$$\lambda^{-1} = \ln(T/T_c) + \sum_{n\leq 0} \frac{1}{n+1/2} \tag{2.4}$$

is the coupling constant. Moreover, $\mathcal{Y}$ denotes the superconducting symmetry basis function, which for s,d-wave are

$$\begin{array}{rcl} \mathcal{Y}_s(\hat{\boldsymbol{p}}_F) & = & 1, \\ \mathcal{Y}_d(\hat{\boldsymbol{p}}_F) & = & \sqrt{2}\,\cos(2\theta_{\hat{\boldsymbol{p}}_F}), \end{array} \tag{2.5}$$

respectively. The cut-off frequency ε_c is related to the limit of n in the sum.

The total quasiparticle current is given by

$$\boldsymbol{j}(\boldsymbol{R}) = -2\pi e v_F N_F k_B T \sum_{\varepsilon_n \leq \varepsilon_c} \langle \hat{\boldsymbol{v}}_F \cdot g(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\varepsilon_n)\rangle_{\hat{\boldsymbol{p}}_F}. \tag{2.6}$$

The free energy according to the free energy functional by Eilenberger [27] is

$$\delta\Omega = N_F \int d\boldsymbol{R} \left\{ |\Delta(\boldsymbol{R})|^2 \ln\frac{T}{T_c} + 2\pi k_B T \sum_{\varepsilon_n<\varepsilon_c} \left[\frac{|\Delta(\boldsymbol{R})|^2}{\varepsilon_n} - \langle \mathcal{I}(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\varepsilon_n)\rangle_{\hat{\boldsymbol{p}}_F} \right] \right\} \tag{2.7}$$

where

$$\mathcal{I} = \frac{\Delta^* f + \Delta\tilde{f}}{1+ig} \tag{2.8}$$

The local density of states (LDOS) is given by

$$N(\boldsymbol{R},\epsilon) = -N_F \cdot \mathrm{Im}\left[\langle g(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\epsilon)\rangle_{\hat{\boldsymbol{p}}_F}\right], \qquad \epsilon = \varepsilon + i\delta^+ \tag{2.9}$$

And finally, the spectral current density is

$$\boldsymbol{j}(\boldsymbol{R},\epsilon) = -e v_F N_F \cdot \mathrm{Im}\left[\langle \hat{v}_F \cdot g(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\epsilon)\rangle_{\hat{\boldsymbol{p}}_F}\right], \qquad \epsilon = \varepsilon + i\delta^+ \tag{2.10}$$

The expressions for all implemented properties, and in which units they are given, are listed in section 4.3.2.

One method to solve the Eilenberger equation under the normalization condition

is to introduce the parameterization of $\hat{g}$ into the so called coherence functions $\gamma, \tilde{\gamma} \in \mathbb{C}$, which for a spin singlet system results in

$$\hat{g} = -\frac{i\pi}{1+\gamma\tilde{\gamma}} \begin{pmatrix} 1-\gamma\tilde{\gamma} & 2\gamma \\ -2\tilde{\gamma} & -1+\gamma\tilde{\gamma} \end{pmatrix} = \begin{pmatrix} g & f \\ \tilde{f} & \tilde{g} \end{pmatrix} \tag{2.11}$$

Reinserting $\hat{g}$ in coherence function form (2.11) into equation (2.1) gives a set of two uncoupled ordinary differential equations (ODE), one each for γ and $\tilde{\gamma}$. This is known as the Riccati formulation [31, 32] of the problem, since the resulting equations can be written on Riccati form as

$$\partial_x \gamma = q_0 + q_1\gamma + q_2\gamma^2, \tag{2.12}$$

for which there exists analytical solutions under certain restrictions. In our case it means the superconducting order parameter Δ has to be constant in space. However, this restriction can be circumvented by solving the equations in small intervals, together with boundary value matching. The details are covered in the following sections.

2.1 Riccati equations

As a result of the parameterization of $\hat{g}$ according to (2.11) we can rewrite (2.1) as a pair of ODEs, in this context simply called the Riccati equations. For singlet pairing, excluding self energies, these are

$$\begin{aligned} (i\boldsymbol{v}_F \cdot \nabla + 2\epsilon)\gamma &= -\Delta^*\gamma^2 - \Delta \\ (i\boldsymbol{v}_F \cdot \nabla - 2\epsilon)\tilde{\gamma} &= -\Delta\tilde{\gamma}^2 - \Delta^*, \end{aligned} \tag{2.13}$$

where $\epsilon \equiv i\varepsilon_n$ for Matsubara frequencies, or $\epsilon \equiv \varepsilon \pm 0^+$ for real energies. If an induced or external magnetic field is to be included, it enters through an energy shift by the vector potential $\boldsymbol{A}$. The energy term then transforms as $\epsilon \to \epsilon + \boldsymbol{v}_F \cdot \frac{e}{c}\boldsymbol{A}$. Numerically, the equation for γ ($\tilde{\gamma}$) need to be solved along the positive (negative) direction of the Fermi velocity, as implied by the gradient term. The reason for the opposite direction of integration for $\tilde{\gamma}$ is related to the particle hole symmetry. Thus, we parameterize these equations along trajectories $\boldsymbol{r}(x) = \boldsymbol{r}_0 + x\hat{\boldsymbol{v}}_F$ for γ, and $\tilde{\boldsymbol{r}}(x) = \tilde{\boldsymbol{r}}_0 - x\hat{\boldsymbol{v}}_F$ for $\tilde{\gamma}$, where $\boldsymbol{r}_0$ and $\tilde{\boldsymbol{r}}_0$ are the intersections of corresponding trajectory with the integration domain boundary (Figure 2.1). The resulting equations can then be written in 1-dimensional form as

$$\begin{aligned} (+i\partial_x + 2\epsilon)\gamma &= -\Delta^*\gamma^2 - \Delta \\ (-i\partial_x - 2\epsilon)\tilde{\gamma} &= -\Delta\tilde{\gamma}^2 - \Delta^*, \end{aligned} \tag{2.14}$$

where the units of the model are expressed in coherence lenghts $\xi_0 = \hbar v_F / k_B T_c$ for distance, and energies in terms of $k_B T_c$. Since we need to perform the integral in (2.3) over the Fermi surface, equations (2.13) need to be reparameterized for

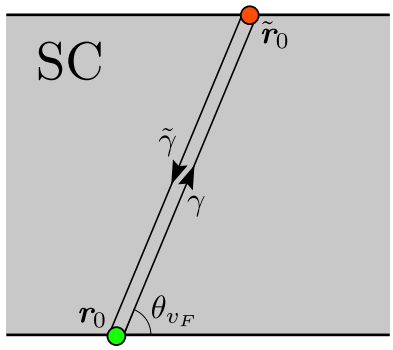

Figure 2.1: *Start point ($\boldsymbol{r}_0$, green) and end point ($\tilde{\boldsymbol{r}}_0$, red) for the γ trajectory where the angle between $\boldsymbol{v}_F$ and the a-axis is θ_{v_F}. Conversely, the corresponding trajectory for $\tilde{\gamma}$ starts at $\hat{\boldsymbol{r}}_0$ and ends at $\boldsymbol{r}_0$. (Note that both trajectories actually overlap in space. The offset between γ and $\tilde{\gamma}$ is only introduced for the viewer to be able to distinguish between the two trajectories.)*

each discrete momentum direction. The form of (2.14) can be kept, but $\boldsymbol{r}_0$, $\tilde{\boldsymbol{r}}_0$ and $\hat{\boldsymbol{v}}_F$ need to change correspondingly. A formal treatment of this technique is described in [11]. An in depth description of how to solve these equations for a discrete 2-dimensional system is covered in section 4.1.

Riccati formalism states that an equation written on the form

$$\partial_x \gamma = q_0(x) + q_1(x)\gamma + q_2(x)\gamma^2 \tag{2.15}$$

has the following solutions

$$\gamma = \gamma_h + \frac{1}{z(x)} \tag{2.16}$$

where γ_h is known (the particular solution), and $z(x)$ is a solution to

$$z' + [q_1(x) + 2q_2(x)\gamma_h]z = -q_2(x) \tag{2.17}$$

To find $z(x)$, we rearrange (2.14) to the form of (2.15), and identify the q-terms. Equation (2.17) then becomes

$$z' + [2i\epsilon + 2i\Delta^*\gamma_h]z = -i\Delta^*, \tag{2.18}$$

if we assume Δ is constant. Solving this equation (see Appendix A for a detailed derivation), we can write the full expression for γ as

$$\gamma(x) = \gamma_h + \frac{2i\Omega C e^{-2\Omega x}}{1 - \Delta^* C e^{-2\Omega x}} \tag{2.19}$$

where C is

$$C = \frac{\gamma(x=0) - \gamma_h}{2i\Omega + \Delta^*(\gamma(x=0) - \gamma_h)} \tag{2.20}$$

Solving for $\tilde{\gamma}$ is analogous (Appendix A), and the corresponding solution is

$$\tilde{\gamma}(x) = \tilde{\gamma}_h + \frac{2i\Omega\tilde{C}}{e^{2\Omega x} + \Delta\tilde{C}} = \tilde{\gamma}_h + \frac{2i\Omega\tilde{C}e^{-2\Omega x}}{1 + \Delta\tilde{C}e^{-2\Omega x}} \tag{2.21}$$

with $\tilde{C}$ as

$$\tilde{C} = \frac{\tilde{\gamma}(x=0) - \tilde{\gamma}_h}{2i\Omega - \Delta(\tilde{\gamma}(x=0) - \tilde{\gamma}_h)} \tag{2.22}$$

2.2 Numerical solution

The expressions for γ (2.19) and $\tilde{\gamma}$ (2.21) are only valid for a constant value of the superconducting order parameter Δ. In contrast, the systems we wish to solve for will have a general form of $\Delta(\boldsymbol{r})$. The solution is to solve the Riccati equations piecewise analytically, by assuming a constant Δ over a short distance h. A good choice for h is, naturally, the lattice spacing.

Starting at the boundary, some value γ_b is prescribed. For bulk solutions $\gamma_b = \gamma_h$. The term $\gamma(0)$ in the equations is to be interpreted as the value of γ at the starting point of integration. To integrate one step from discrete coordinate j, i.e. to find the value of γ_{j+1}, given γ_j, the expressions for the coherence functions can be interpreted as follows

$$\gamma_{j+1} = \gamma_h + \frac{2i\Omega_n C_n}{e^{2\Omega_j h} - \Delta_j^* C_j} = \gamma_h + \frac{2i\Omega_j C e^{-2\Omega_j h}}{1 - \Delta_j^* C_j e^{-2\Omega_j h}} \tag{2.23}$$

where

$$C_j = \frac{\gamma_j - \gamma_h}{2i\Omega_j + \Delta_j^*(\gamma_j - \gamma_h)} \tag{2.24}$$

$$\gamma_h = -\frac{\Delta_j}{i\varepsilon_n + \boldsymbol{v}_F \cdot \boldsymbol{A}(\vec{x}) + i\Omega_j} \tag{2.25}$$

$$\Omega_j = \sqrt{|\Delta_j|^2 - (i\varepsilon_n + \boldsymbol{v}_F \cdot \boldsymbol{A}(\vec{x}))^2} \tag{2.26}$$

where n indicates Matsubara frequency, $\vec{x}$ is the 'global' spatial coordinate (the coordinate in the reference frame), $\boldsymbol{v}_F$ is the Fermi velocity which coincides with the direction of integration for γ, and h is the physical spacing between 2 adjacent discrete grid points. Functionally, the dependencies in the computation above can, somewhat simplified, be expressed as $\gamma_{j+1} = F[\gamma_j, \Delta_j]$. When γ_{j+1} is computed, the next discrete value of the solution is given by $\gamma_{j+2} = F[\gamma_{j+1}, \Delta_{j+1}]$, and so

on until we reach the end of the trajectory.

For $\tilde{\gamma}$ the procedure is analogous. However, the integration starts from the end point of the γ trajectory and integrated along the opposite direction. So to compute $\tilde{\gamma}_{j-1}$, the discrete values $\tilde{\gamma}_j$ and Δ_j are used. For the boundary value $\tilde{\gamma}(x = 0)$ in (2.22), the coordinate is referring to the parameter along the integration trajectory of $\tilde{\gamma}$. This coordinate is the same as the end point of the γ trajectory, as illustrated in Figure 2.1. In a global coordinate system, the substitution $\gamma(x = 0) \to \gamma(\boldsymbol{r}_0)$ and $\tilde{\gamma}(x = 0) \to \tilde{\gamma}(\tilde{\boldsymbol{r}}_0)$ have to be made. But, $\boldsymbol{v}_F$ in the expressions for $\tilde{\gamma}$ is the same as in those for γ.

The integration "recipe" described above provides a fast and stable solution method. There are of course other numerical methods of solving the Riccati equation. One popular method often used for initial value differential equations is 4th order Runge-Kutta. This can also be applied to our problem, but is left for a future implementation.

2.3 Self-consistent iterations

From equations (2.3) and (2.11) we see that there is a functional dependence for the order parameter $\Delta = F[\gamma,\tilde{\gamma}]$. Conversely, in the Riccati equations (2.13) the reversed dependence is found where $\gamma,\tilde{\gamma} = G[\Delta]$. For the general case, we cannot write an expression for Δ in closed form. Thus, to find a solution we must iterate the computation of Δ until self-consistency is reached. The algorithm is simple (but time consuming); we start with an initial guess for $\Delta = \Delta_0$, from which we can compute γ and $\tilde{\gamma}$. Now a new order parameter can be computed as $\Delta_1 = F[\gamma,\tilde{\gamma}]$. We update $\Delta \leftarrow \Delta_1$, and repeat the procedure. When some measure of the difference between two iterations is small enough, we consider the solution to be converged. We use the following measure

$$\delta\Delta = \frac{||\Delta_{i+1} - \Delta_i||}{||\Delta_i||}, \tag{2.27}$$

where the subscript indicates iteration number. The number of iterations to convergence naturally varies depending on the tolerance threshold, but there are also other factors depending on the nature of the system. For example, the edge current phase in paper I requires an unusually large number of iterations to reach convergence. Generally, when the free energy "landscape" is very flat, the number of required self-consistent iterations tend to increase, as the solution can become stuck in local minima for some time. A good initial guess is always recommended to minimize the time to convergence.

Chapter 3

Key Programming Concepts

With the goals and design principles in place, one can now proceed and choose the programming language(s) deemed most suitable for the implementation. There are many candidates indeed, and we chose C++ for the framework itself, and CUDA C for the computational core. The motivation for C++ is that it is both a modern and well established object oriented programming language. It is also computationally efficient, with enough flexibility in the language to allow for all the properties previously discussed. The downside is that it has a bit of a learning curve and can be intimidating to use at first if your only previous programming experience are interpreted languages like MatLab. Python would have been a good choice in terms of initial development speed, but because of performance considerations some of the contents would likely have had to be written in C/C++ anyway. This would then result in three different languages instead of two, and also an extra programming burden to interface the Python and C/C++ code. A possible feature that has been discussed, but not implemented, is to write a Python layer on top of a few key classes in the final C++ library so that the end user only has to use Python when setting up a simulation. This would have the benefit of making the initial learning curve less steep, from a user perspective.

The computationally intensive part, solving the Riccati equations, is a highly parallel task, and therefore it is natural to use CUDA C. This is a programming language which utilizes the computing power in a graphics card (GPU) to perform general computations. An early adoption of this approach for this particular problem was done by Wennerdal in [33]. Modern GPUs harness an impressive computational capacity when compared to a CPU, *if* your problem can be mapped to many ($> \sim 10^3$) individual tasks that can be computed in parallel. As the name suggests, its syntax is closely related to that of C, but has some strict guidelines in how you set up your code in order to attain optimal performance.

Following this chapter is a more in depth description of the programming languages introduced above.

3.1 C++ concepts

C++ started as an extension to the C language, and was first called "C with Classes". It was first developed in 1979 by Bjarne Stroustrup at Bell Labs in an attempt to add more high level programming to C. Later on a standards committee was formed to steer the development of C++, which still operates today. Some of the key concepts supported in C++ through classes include: *abstraction*, *inheritance*, *polymorphism*, and *encapsulation*. All of these are utilized in our code library, so it's instructive to explain them in more detail. But first we must look at what makes a class.

A class in C++ is basically a collection of functions and variables, which are called class members. The purpose is to create a logical grouping which contains all the necessary information and functionality to more easily deal with representations which are more complex than the native data types. The class itself is just a description of the representation, a blue print. To use the class we create instantiations, or objects, and assign values. A simple example would be the representation of a 3-dimensional vector, which would contain storage of the coordinates as well as functions to provide for vector operations. To create two vectors, we instantiate two objects and give them coordinates. Vector operations can then be applied to these, according to the functions defined in the class description. This way the class provides an **abstraction** layer for a vector, meaning we do not have to worry about implementation details or the specific mathematics of different vector operations. A more complex example of a class could be a geometry representation, where e.g. surface properties can be queried and modified. Each instantiation would then represent a different shape.

The idea of **encapsulation** is to hide implementation details and data from the user to ensure it is used in a correct manner, as defined by the class member functions and operators. If unrestricted access is allowed to a complex data structure, a user could accidentally modify the data, or try to modify it in a way which renders it invalid. There can also be data dependencies between different class member variables. Directly changing one variable could break these dependencies and the behavior will no longer be correct. Encapsulation ensures that we can not incorrectly change the data, intentionally or unintentionally. It also simplifies usage, since only the necessary controls are exposed.

Inheritance is the property where a new class can inherit all the properties from a previously defined class. This can prove to be very convenient, but equally important is that it also has a conceptual meaning. One can implement a class which deals with all the generic functionality of a geometry. From this *base* class, we can create a circle class which inherits from the geometry class. The circle class is called a *derived* class, and the only extra work we need to implement is all the features unique to a circle. Conceptually, this has the implication that we can now say that while the circle is a circle, it is also a geometry. As a consequence, everywhere in the code where a geometry is accepted as input, a circle will also be accepted.

With **polymorphism** one generally refers to the ability of an object to behave in different ways, depending on context. This can be accomplished in a few different ways in C++. The perhaps simplest realization of polymorphism is by *function overloading*. Here, several instances of a function with the same name can be implemented, but with varying number of input parameters and for different types. This way you can get a different behavior depending on the arguments used. Another way in which polymorphism can be exhibited is by *virtual inheritance*. By declaring a function in a base class virtual, it is always the most specialized version (in derived classes) of this function which is called. For example, say that we mark the member function Geometry::area() as virtual. We then write specialized versions of area() for two derived classes Circle and Rectangle. As mentioned earlier, a function which accepts the Geometry class will also accept any derived classes. However, inside the function the input argument is just known as a Geometry object, the derived type which we called it with is unknown. But when the area() member function is called from the input object, it will call Circle::area() if we called the function with a Circle object, and analogously for a Rectangle object. The following code demonstrates this property:

```
void func(Geometry geom)
{
  print geom.area();
}

Circle myCircle;
Rectangle myRectangle;

func(myCircle);     // Calls myCircle.area()
func(myRectangle);  // Calls myRectangle.area()
```

3.1.1 C++ Templates

One can also achieve polymorphism by a feature in C++ known as *templates*. The use of templates can be very versatile, but can in its simplest form be viewed as a way of allowing for more flexibility in a very strongly typed language. Consider the case where we want to implement a sort function. Without templates, the entire function would have to be re-written for every data type (int, float, double, short, Circle, etc) we want it to be able to sort, even though the algorithm is the same. When employing templates, the function only have to be written once. One way to look at it is to consider the data type itself as a variable, formally called a template argument. Following is a listing showing how a (mostly pointless) multiplication function template would look like:

```
// Define the function
template <typename TYPE>
TYPE mult( TYPE a, TYPE b )
{
  return a*b;
}

// Use the function with any datatype which
// has the operator * defined
int a = 5;
int b = 3;
int c = mult<int>(a,b);

float d = 5.1;
float e = 3.1;
float f = mult<float>(d,e);
```

Templates allow for *generic programming*, where one can focus on design patterns and algorithms without having to consider data types and the underlying implementation. A very good example is the Standard Template Library [34], which is used extensively and now part of the C++ standard.

3.2 GPU Computing

A graphics card is, as the name suggests, a separate unit whose purpose is to handle graphics operations. The idea is to offload the CPU of these tasks in order to lessen its computational burden. The nature of these graphics operations are usually such that they can be computed in parallel. Because of the limited scope of these operations, it is sufficient with a less advanced microarchitecture than a CPU. As a solution, the graphics processing unit (GPU) was developed. It compensates the reduced complexity with significantly increased raw computational power. It was soon discovered that many problems in computational physics could be mapped to a series of simpler graphics operations [35]. What they had in common was that both had a large amount of data to be processed with the same mathematical operation. It was simply a matter of handing a large array of data to the GPU, packaging it as an image. This approach was cumbersome for more complex problems, however. But the GPU manufacturers, among others, had realized by now the potential for the GPU to solve general problems in computational science. Consequently, programming languages for the GPU emerged, behaving more like traditional programming languages like C, and offered more control than was possible with regular graphics operations. The two most most widely adopted languages today are CUDA C and OpenCL, both of which offer C like syntax. CUDA C is a proprietary language developed by Nvidia Corporation, and can only be used with Nvidia hardware. OpenCL is developed by the

non-profit consortium Khronos Group, consisting of around 100 member groups including Intel Corporation, ATI technologies, Nvidia, Silicon Graphics and Sun Microsystems. The goal with OpenCL is to provide an open standard to heterogeneous computing, where one can dispatch the computational work to a large number of platforms, including CPUs and GPUs, independent of manufacturer, and even certain mobile phones. The most widely used language in academia and industry is arguably Nvidias CUDA C, since it is generally considered to be more developed and productive than OpenCL, offering a larger degree of functionality and control.

It is instructive to compare the numbers of a CPU and a GPU. A modern high end server CPU, such as Intel Xeon E5-2697v2, has 12 computational cores, each capable of performing independent work in parallel with the other cores. The peak bandwidth is around 60 GB/s, meaning the maximum rate at which it can read data from memory, and its theoretical maximum number of floating point operations per second (FLOPS) is around $260 \cdot 10^9$ FLOPS (260 GFLOPS) [36]. A GPU in the same price range, like Nvidias Tesla K20x, can process 2688 parallel operations at once, has a peak bandwidth of 250 GB/s, and a theoretical maximum of 1.3 TFLOPS. In reality, it is impossible to give a single number saying how much faster the GPU is, since it depends very much on the nature of the problem, but it commonly seems to be in the range of a factor of 5-10x speedup [37]. The fact that the Tesla card has ~ 200 times more cores unfortunately does not mean we can get that kind of speedup. What it does mean is that for the GPU to be efficient, the problem needs to be divisible into a large number of individual tasks that can all be computed independently of each other. Also noteworthy is that the GPU power consumption is somewhat less than double that of the CPU, meaning the GFLOPS/Watt ratio is much higher for the GPU, i.e. more energy efficient.

In the vast majority of cases with GPU computing the GPU uses its own dedicated memory, which is not directly accessible from the CPU. The reason for not sharing memory is primarily twofold. Firstly, the graphics card is a peripheral extension, which is connected to the PCI bus of the computer. The bandwidth over the PCI bus is very low in this context at ~ 10 GB/s. As a consequence, the maximum computational capacity of the GPU would be severely throttled if all data would have to move across the PCI bus, rendering its use completely pointless. Instead a local memory is installed, very closely connected to the GPU chip. This memory (most often of the type GDDR5) is more than an order of magnitude faster than the transfer rate of the PCI bus. The local memory is also almost an order of magnitude faster than what can be sustained by a high end CPU, and this relates to the second reason why a special memory is employed. In principle, one could hardwire the GPU to the motherboard to avoid the PCI bus, having it share memory with the CPU. But the way a GPU accesses memory (in efficient code) is more specific than that of a general purpose CPU, and therefore the memory can be optimized specifically for this usage. Memory has two main measures of performance, latency and bandwidth. Latency is the time which it

takes to initiate a random memory access, while bandwidth is the maximum rate of data throughput once initial access has taken place. A CPU has low latency and low bandwidth, compared to GPU memory. This is because a CPU needs to have good all round performance, and perform all sorts of tasks not tied to any particular memory access pattern. In contrast, GPU memory has much higher bandwidth, at the cost of a significantly higher latency. This trade-off is made on the assumption that the data will typically be accessed in a predictable sequential fashion. It is now easy to see that applications requiring a random access pattern in memory is not suitable for a GPU. The optimal case is one where large chunks of data can be read sequentially, which is also true for a CPU, but the difference in performance between the two access patterns is much more pronounced for a GPU because of the high latency.

3.3 CUDA programming model

The GPU programming language used in this thesis project is CUDA C. Although it is desirable to use an open standard like OpenCL, available to a large number of platforms, our reasoning is that the CUDA framework facilitates faster development, has more advanced features, and can be more tightly integrated with C++. For example, the use of templates is an important feature which is exclusive to CUDA C. Following is an overview of the CUDA programming model. To distinguish between code and data residing on the CPU- or GPU-side, we adopt the widely used terminology of host (CPU) and device (GPU), as this is more natural to use in writing (and speech). Worth mentioning is that many of the concepts are very similar in OpenCL, just with a different nomenclature.

The modus operandi of GPU-computing is to have a large set of data upon which the same operation(s) is applied, also known as SIMD (Single Instruction, Multiple Data). The total amount of work to be computed is partitioned and dispatched to be processed as individual tasks, all computed in parallel. One such individual task, or process, is called a *thread* in CUDA C. Since the maximum number of simultaneously active threads varies between different existing hardware, and will likely vary in the future, a partitioning scheme is built in to the language. This allows the total work to be split up into blocks of work, known as *thread blocks*, which can be mapped efficiently to hardware with different capabilities. Together, all thread blocks form what is called a *grid*, whose dimensions are equal to the total problem size. See figure 3.1 for an illustration of the thread grouping hierarchy. In order to map the threads intuitively to the problem, a thread block can be logically arranged as a 1-, 2-, or 3-dimensional entity. Each thread is then endowed with an index in each dimension specifying its ”location”. If the computational problem involves a large matrix, a 2-dimensional structure would be preferential. The thread index $(x,y) = (4,8)$ could then map to the corresponding (row,column) entry in the matrix (note that for an efficient implementation, this particular mapping requires the matrix to be stored column-major

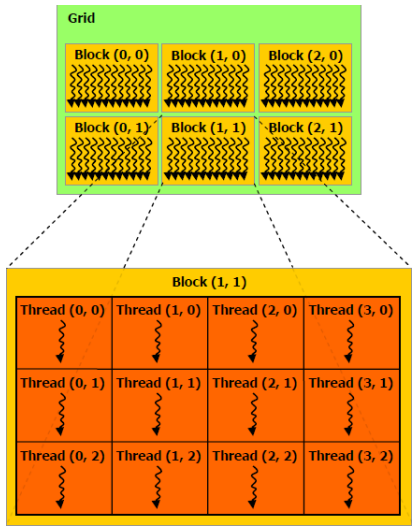

Figure 3.1: *CUDA thread model. The grid consists of a number of blocks, and each block is made up of a collection of threads. Each thread is capable of performing individual work. The logical arrangement of these structures can be 1-, 2-, or 3-dimensional in order to more intuitively map the threads to the problem.*

in memory). One could of course construct a 1-d thread block which would be equally computationally efficient, but requiring more code to find the item to be processed. Analogously, the thread blocks themselves can also be assigned a dimensional property.

The code which gets executed on the device is put in a special function called a *kernel*, which in turn is called from the host. Since the mapping of the problem onto threads is done through the dimensioning of the thread blocks, we don't have to loop over data items as one would have to in sequential code in e.g. C/C++ or Fortran. The following two code snippets computing the sum of two 1-d vectors are functionally equivalent

```
// C/C++
for ( int i=0; i<N; ++i )
{
  c[i] = a[i] + b[i];
}

// CUDA C
int i = blockIdx.x * blockDim.x + threadIdx.x;
c[i] = a[i] + b[i];
```

For a vector of N elements, and with a 1-d thread block of 64 threads, the kernel call from the host side would have the following form

```
vec_add<<<N/64,64>>>(a,b,c);
```

Inside the special brackets `<<<>>>` the thread block and grid dimensions are specified. The first entry defines how many thread blocks the grid is comprised of, and the second entry the number of threads in a thread block. The thread partitioning is completely analogous in the 2- and 3-dimensional cases.

3.3.1 Warps and coalescent access

The hardware architecture is designed such that the threads operate simultaneously in groups of 32. Such a group is called a *warp*. It is therefore optimal to construct thread blocks in integer multiples of warps, and never smaller than a warp. If the problem were to be divided into thread blocks of 33, this would lead to two warps (64 threads) being scheduled for each thread block, but with only 33 active threads, effectively cutting performance in half since nearly half of the threads would always be idle.

Similarly to how threads operate in warps, data from memory is always read in blocks of 128 bytes. To attain optimal performance, memory reads should, when possible, be aligned with these blocks. Say each thread in a warp reads one `float` (4 bytes) from global memory, which is a total of 128 bytes for the entire warp. If these data elements are stored in one contiguous block in memory, all data can be fetched in one read. Such an access is called *coalesced*. If the data each thread wants is scattered all across memory, 32 separate data reads have to be performed. This will incur a major performance drop, since as a result, 4096 bytes of data have to be read. In light of this, it becomes obvious that we must carefully think about how data is partitioned in memory. As an illuminating example, consider an array of complex valued numbers for which we want to compute the complex magnitude, where each thread handles one complex number. Each thread will then have to do two reads from the global memory: the real and imaginary part, one at a time. If the data is laid out in memory such that each (Re,Im) pair are next to each other, the memory access will have the pattern illustrated in Figure 3.2 (top). Here the access is not coalesced, and half the bandwidth (performance) will be wasted. If the data is instead structured such that the real and imaginary components are stored in separate arrays, the result is two coalesced memory reads (Figure 3.2(bottom)), and all the bandwidth is utilized.

3.3.2 GPU memory structure

There are different types of memory available to the GPU. These can roughly be divided into what is known as global memory and shared memory. The global memory is large and comes with high latency, as discussed earlier. If one thread needs to access data many times, or more than one data item, this can incur a

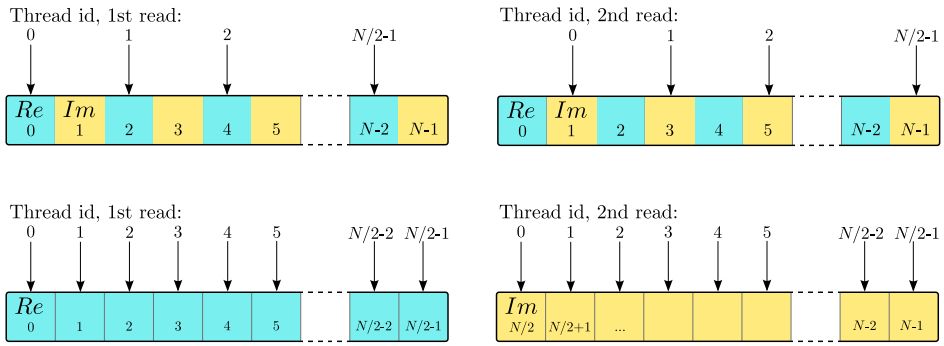

Figure 3.2: *Memory access pattern for a collection of CUDA threads reading a complex valued array. Top: If the real and imaginary components are stored together (a pattern known as array-of-structures), the memory access will not be coalesced and half the bandwidth will be wasted. Bottom: By storing the real and imaginary components in separate arrays (structure-of-arrays), the access is coalesced and the utilization of bandwidth is optimal.*

severe performance penalty. As a remedy, the shared memory is introduced. This is a small, fast low latency memory, local to each thread block, where data can be temporarily stored and accessed in any order with little performance impact. Moreover, all threads inside a thread block can now exchange data with each other through the shared memory, hence the name. There is also private memory with the same performance as the shared memory, but, as the name implies, data in this memory cannot be shared between threads (without going through global memory).

Chapter 4

Implementation

In this chapter a detailed description is given of how the Riccati equations are solved on a discrete 2-dimensional lattice. We also explain how to maintain a continuous boundary description on a discrete lattice, and how to implement the specular boundary condition. Moreover, the computational framework is introduced, where the key functionality of the API is presented, i.e. what types of systems the solver is capable of, and which physical properties it can compute. Furthermore, an overview of the key components (classes) of the API is given, with the purpose of introducing the reader to the framework and its structure. Finally, the complete code for setting up a system identical to the one studied in paper I is listed, where each operation is thoroughly commented.

4.1 Solving the Riccati equations in 2D

As described in section 2.1, the Riccati equations for γ ($\tilde{\gamma}$) are parameterized along the positive (negative) Fermi velocity. For brevity, the following description will treat γ only. However, the procedure for $\tilde{\gamma}$ is analogous, with the exception that the starting point of integration is at the end point of the γ trajectory, and proceeds in the direction of the negative Fermi velocity, ending at the starting point of the γ trajectory (Figure 2.1).

By assuming Δ to be constant under a short interval h, the equations can be solved piecewise, starting at $x = 0$ and finishing at the end of the solution domain where $x = L$. When solving for the coherence functions, we need the value of Δ at the discrete points $x = 0$, h, $2h$, ... to evaluate the expression (2.23). When $\hat{\boldsymbol{v}}_F$ is aligned with the cartesian axes, we can choose to solve the coherence functions at the exact same lattice points at which Δ is defined (Figure 4.1). However, for all other cases it is impossible to have the coherence function lattice line up with the lattice of Δ (Figure 4.2). One solution is to interpolate Δ from the nearest neighbors. This of course gives an approximation, but as long as the gradients are

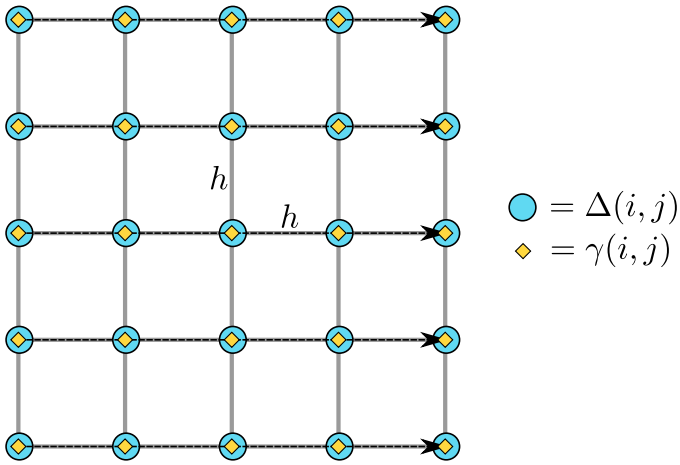

Figure 4.1: *When the Riccati equations are integrated along any of the cartesian axes, the discrete solution (yellow diamonds) can be chosen to coincide with the discrete lattice points at which the order parameter is defined (blue circles). The dashed arrows indicate direction of integration.*

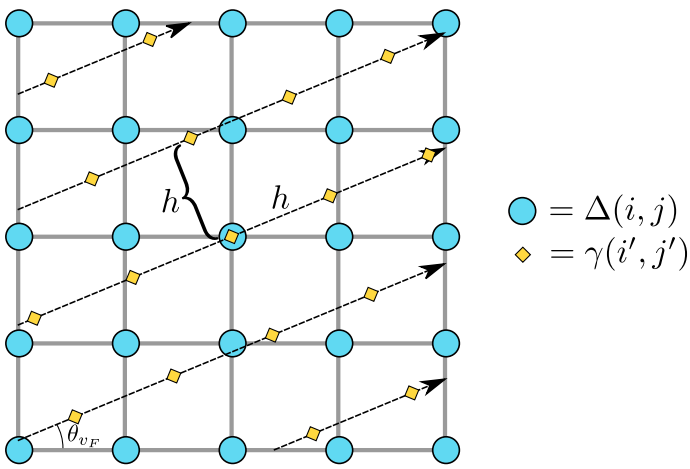

Figure 4.2: *In general, the Riccati equations are not integrated along a cartesian axis, and we can not choose a lattice for the Riccati solution (yellow diamonds) which coincides with the order parameter lattice (blue circles).*

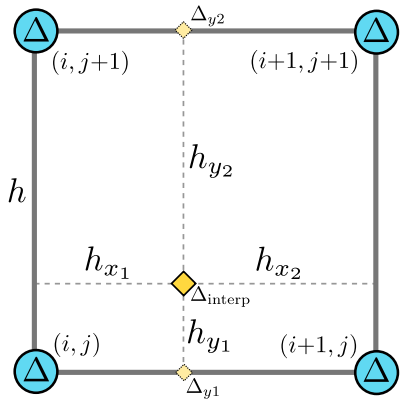

Figure 4.3: *To obtain a value of the discrete lattice order parameter Δ at an arbitrary coordinate, bilinear interpolation is used where the approximated value Δ_{interp} is determined by the 4 nearest neighbors.*

small, which can be accomplished by choosing a sufficiently high discrete lattice resolution, the error will be small. In this solver, bilinear interpolation is used, where Δ is approximated at any given point in space by using the 4 nearest neighbors. With the notation from Figure 4.3, the interpolated value is given by

$$\Delta_{y1} = \frac{h_{x_2}}{h}\Delta(i,j) + \frac{h_{x_1}}{h}\Delta(i+1,j) \tag{4.1}$$

$$\Delta_{y2} = \frac{h_{x_2}}{h}\Delta(i,j+1) + \frac{h_{x_1}}{h}\Delta(i+1,j+1) \tag{4.2}$$

$$\Delta_{\text{interp}} = \frac{h_{y_2}}{h}\Delta_{y1} + \frac{h_{y_1}}{h}\Delta_{y2} \tag{4.3}$$

However, as a consequence the coherence functions are now computed at the same points that were not part of the discrete lattice in the reference frame. Again, using the same interpolation technique we can approximate the values of the coherence functions in the points where Δ is defined.

In practice, this is implemented by pre-computing all the interpolated values for Δ and storing them in a lattice which coincides with the chosen one for the coherence functions. This is essentially a rotation of the order parameter. Once all the coherence functions have been solved for a particular momentum in the momentum integral, the desired Green's functions are computed. These are then interpolated back (counter rotated) to the *reference lattice* and added to the contribution to the total integral. Thus, the coherence functions are always solved in the same direction locally; it is the rest of the system that is counter rotated (Figures 4.4, 4.5). Solving the Riccati equations along some θ_{v_F} is equivalent to rotating the rest of the system $-\theta_{v_F}$ and solving in local frame $\theta = 0$. The

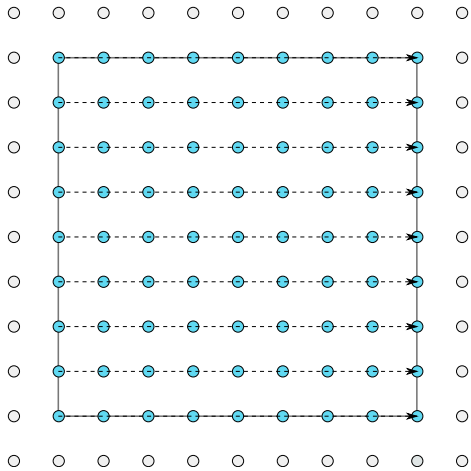

Figure 4.4: *Integration of the Riccati equations in the reference frame, i.e. when* $\theta_{v_F} = 0$. *The gray circles represent discrete lattice points outside of the solution domain. The blue circles represent the lattice points where* Δ *is defined.*

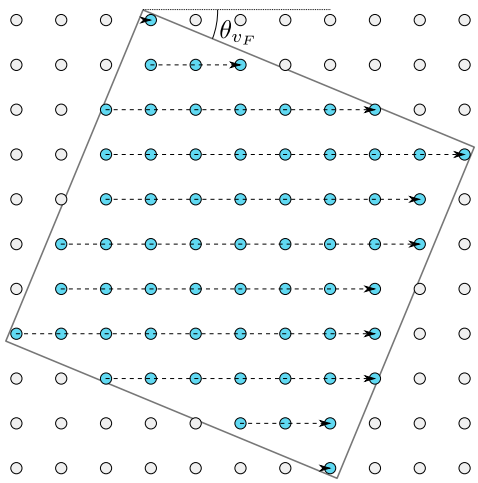

Figure 4.5: *Integration of the Riccati trajectories for a general momentum direction, or rotational frame, where* $\theta_{v_F} \neq 0$. *Here, the order parameter is counter-rotated* $-\theta_{v_F}$, *which is equivalent to a pre-computed interpolation. Thus, the blue circles now represent interpolated values of* Δ. *The Riccati trajectories can now be integrated along the lattice in local frame* $\theta' = 0$. *When the integration is completed, the result is rotated* $+\theta_{v_F}$ *and interpolated onto the reference frame lattice.*

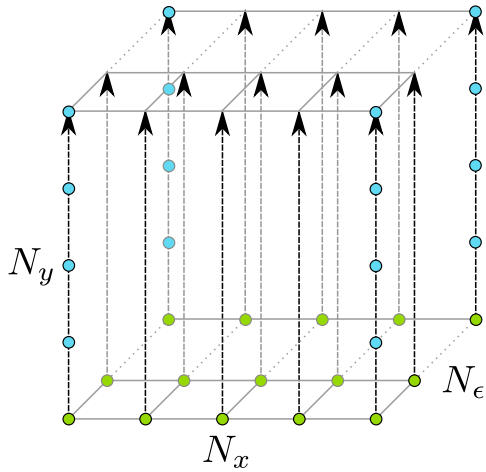

Figure 4.6: *The structure of the CUDA thread grid used for the kernel responsible for solving the Riccati equations. The circles in general represent the discrete lattice. The green circles are the launched threads. Each line/arrow represent the work done by a single thread. The decision to integrate along the y-axis is for reasons of computational efficiency. This way coalesced memory access is achieved. The grid is a slice in the* (x,ϵ)*-plane because all the trajectories along the x-axis and all the energies can be computed in parallel. The y-axis has a dependence where* $\gamma(x,y+h;\epsilon) = f[\gamma(x,y;\epsilon)]$*. Thus the values along the y-axis can not be done in parallel.*

algorithm for computing the momentum integral for any Green's function in the matrix $\hat{g} = \hat{g}(x,y)$ is as follows:

- $g \leftarrow 0$
- For each discrete angle θ in [0,...,2π):
  - Transform Δ to local frame: $\Delta' \xleftarrow{\text{Rot } -\theta} \Delta$
  - Solve $\gamma,\tilde{\gamma} = \gamma,\tilde{\gamma}(\Delta')$ in local frame
  - Compute $g'_\theta = g'_\theta(\gamma,\tilde{\gamma})$
  - Transform to reference frame: $g_\theta \xleftarrow{\text{Rot } \theta} g'_\theta$
  - $g \leftarrow g + g_\theta$

For performance reasons, the coherence functions are always solved along the y-axis (in the local frame). Only then can we achieve coalesced memory access with the GPU. The grid of CUDA threads has the layout illustrated in Figure 4.6. The threadblocks are 1-dimensional along the x-axis, while the threadblocks themselves are stacked along the energy-axis. Each thread is then responsible for

integrating along the entire y-axis for its unique (x,ϵ) coordinate. Thus, when each warp grabs a chunk of data from global memory, it will do so along the x-axis, which is the same order as the data is arranged in memory by the `GridData` class. After advancing the solution of the Riccati equation one lattice step along the y-axis, another chunk of data is read, again a coalesced read along the x-axis, and so on until the end of the integration trajectory is reached.

4.2 Specular boundary condition

In the solving methodology outlined above, only bulk systems are concerned, where we look at an arbitrary region in an infinitely large system. At the starting point of a Riccati trajectory, the bulk solution is used as initial/boundary value. To solve the equations for a finite size system, a perfectly clean superconducting grain, the Riccati trajectories are subject to specular reflection at the boundaries (for a general treatment on boundary conditions, see [30]). While mathematically trivial, this boundary condition requires some effort for a computational and resource efficient implementation. The main issues and corresponding solutions are described below.

First, a well defined description of the geometry must be provided, inside which the equations will be solved (also referred to as the solution *domain*). Simply defining a geometry by selecting a certain set of lattice points is ill defined, since the geometry is not defined between the lattice points, and thus the normals are also undefined. Moreover, we would be limited to the geometrical shapes which can be described by the available lattice points, and changing the discrete resolution would effectively change the geometry.

The adopted solution is to define the geometry by shape primitives, each of which is continuously defined (defined everywhere). These shapes need not be constrained to the available lattice points, and are resolution independent. They are also, naturally, well defined in any rotational frame. The shape primitives implemented are a disc and a non-intersecting polygon with any number of vertices, and the full geometry can be constructed by aggregating any number of instances of these shapes. Moreover, the primitives can be used to either add or remove from the shape to be constructed. When implementing this type of geometrical description, there are a number of issues related to the discreteness of the solution lattice which needs to be addressed. There must be an easy way to distinguish whether a given lattice coordinate is inside the defined geometry or not. Also, because the geometry boundary will generally not coincide with the lattice points, we must be able to handle fractional lattice distances. And finally, we must be able to easily find the normals at the exact coordinate where the Riccati trajectory intersects the boundary. The implemented solution is to generate three auxiliary lattices containing the information required. The first lattice labels each lattice point, identifying whether it is outside, inside, or near (less than one lattice distance) the boundary. It also identifies if the boundary is one where the trajectory

enters the domain, or exits it. Figure 4.7 illustrates the information contained in this lattice. The second auxiliary lattice contains the exact distance (in lattice units) to the nearest boundary. With this information the Riccati equations can be solved up to the exact location of the boundary, rather than be limited to the lattice. Finally, the third auxiliary lattice contains the normals at the exact point at which the Riccati trajectory intersects the boundary. The normals are stored at the discrete lattice points which are marked as boundary by the first auxiliary lattice, but when created they were evaluated at the exact boundary location, and by using the boundary distance auxiliary lattice, we can recover the exact point in space at where they belong. These normals are then used when enforcing the specular boundary condition. The method just described addresses the issue of solving the Riccati equations, when starting the trajectory from the exact boundary location and solving up to the exact location of the boundary at which it exits the geometry, as illustrated in Figure 4.8. What remains to be answered is how to choose the starting value of γ at the boundary when entering the geometry. As a result of the specular boundary condition, each commencing trajectory at the boundary will have arrived to that point from a previous trajectory. The starting point for the current trajectory with momentum p_o, was the ending point of a previous trajectory with momentum p_i. With the help of the boundary normals, p_i can be inferred. The question now is, how can we know the value of γ_{p_i} as it terminated its trajectory? The solution employed here is to record and store the values of *all* the trajectories as they terminate at the boundary. To avoid excessive memory usage, this data is stored in a sparse array (see section 4.4 on `BoundaryStorage`). Following is a more detailed description of the discrete boundary treatment.

At the end of each trajectory, the value of the coherence functions are stored at the boundary for each energy/Matsubara frequency, for each discrete angle. As previously mentioned, with knowledge of the boundary normal at the location of the commencing trajectory γ_{p_o}, we can find the momentum p_i of a trajectory terminating at this spatial coordinate. However, the momentum p_i will generally not coincide with the set of existing discrete directions. Therefore an interpolation has to be made. The same reasoning applies to discrete spatial coordinates; the corresponding incoming and outgoing trajectory will not terminate and commence at the exact same spatial coordinate, due to the discreteness of the spatial grid. Figures 4.10, 4.11 illustrate this concept.

In discrete momentum-space, the trajectories that make up the Fermi surface form angles $\Theta_0, \Theta_1, ..., \Theta_{N-1}$ with the x-axis (from here on θ denotes exact angles, while Θ denotes angles from the discrete set). For algorithmical reasons, the outgoing trajectory angle of Γ isn't computed from the incoming γ, but rather the other way around. For a given discrete trajectory Γ with angle $\theta_{out} = \Theta_M$ for some M where $0 \leq M < N$, the *exact* angle θ_{in} of the corresponding incoming trajectory is calculated using θ_{out} and the boundary normal $\hat{n}$ (see figure 4.9). As mentioned before, θ_{in} will in a general case not coincide with a discrete angle

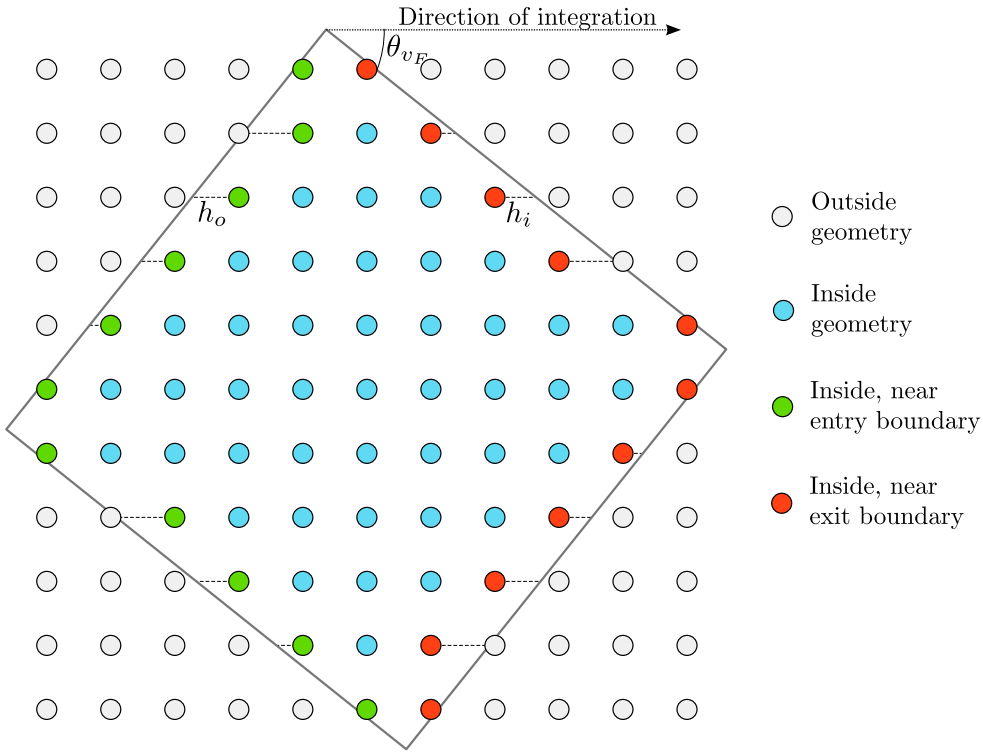

Figure 4.7: *Augmented discrete representation of the geometry. The rotated square is the formal description of the finite size geometry. To supplement the incomplete lattice representation, a few auxiliary lattices are constructed. One contains information stating if a particular lattice point is outside, inside, or near a boundary. Another holds the distance from the current lattice point to the exact geometry boundary. The distance data is stored on the boundary lattice points only, i.e. the green and red circles in this figure. With this additional information, the Riccati equations can be solved for the entirety of the geometry interior, instead of being limited to the lattice approximation.*

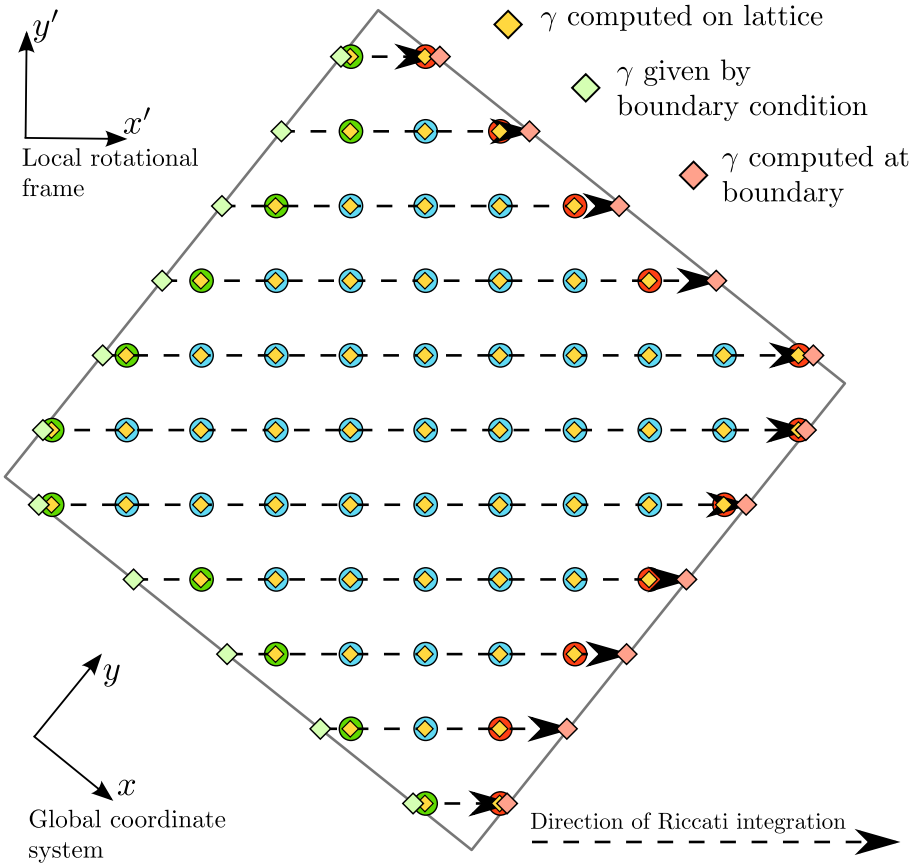

Figure 4.8: *Using the augmented discrete geometry representation (Figure 4.7) to integrate the Riccati equations between the exact limits of the boundary. The initial values of the coherence functions (green diamonds) are retrieved by evaluating the specular boundary condition on the values stored at the exiting boundary (red diamonds).*

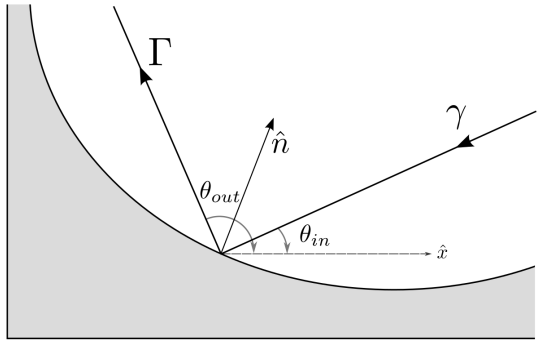

Figure 4.9: *Specular reflection of coherence function in the ideal case (continuous variables).*

Θ, so to find Γ we have to interpolate from the values of γ_{Θ_m} and $\gamma_{\Theta_{m+1}}$, where $\Theta_m \leq \theta_{in} < \Theta_{m+1}$. That is, $\Gamma_{\Theta_M} = a\gamma_{\Theta_m} + b\gamma_{\Theta_{m+1}}$ where $a+b=1$ (figure 4.10). The weights a,b are chosen using linear interpolation:

$$\begin{aligned} a &= \frac{\Theta_{m+1} - \theta_{in}}{\Theta_{m+1} - \Theta_m} \\ b &= 1 - a \end{aligned} \tag{4.4}$$

Furthermore, as discussed previously, an interpolation in real space of γ has to be made for each of the two discrete angles Θ_m and Θ_{m+1} such that

$$\begin{aligned} \gamma_{\Theta_m} &= \alpha_m \gamma_{\Theta_m, x_k} + \beta_m \gamma_{\Theta_m, x_{k+1}} \\ \gamma_{\Theta_{m+1}} &= \alpha_{m+1} \gamma_{\Theta_{m+1}, x_k} + \beta_{m+1} \gamma_{\Theta_{m+1}, x_{k+1}} \end{aligned} \tag{4.5}$$

where

$$\begin{aligned} \alpha &= \frac{x_{k+1} - x_{in}}{x_{k+1} - x_k} \\ \beta &= 1 - \alpha \end{aligned} \tag{4.6}$$

where the k subscript denotes discrete spatial coordinates, and x_{in} is the exact spatial coordinate of the outgoing trajectory Γ. See figures 4.11, 4.12. The effective result of this interpolation is that the scattering becomes slightly diffuse. As the number of discrete angles/momenta are increased, this artificial diffusivity is reduced.

4.3 Framework functionality

In light of the goals set at the beginning of the project (section 1.1), an API (Application Programming Interface) has been developed. An API is a set of

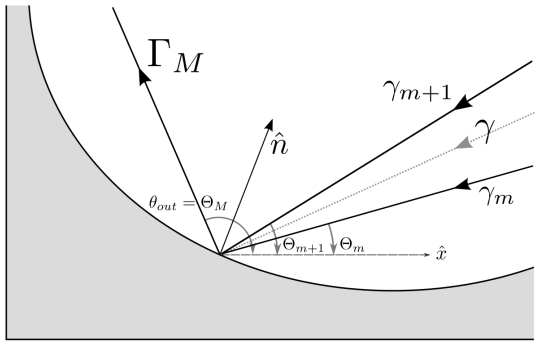

Figure 4.10: *Specular reflection in discrete momentum space. The discrete momenta will in general not coincide with the exact momentum of a reflected trajectory. Lower case subscripts denote indices of incoming trajectories, while upper case denote corresponding outgoing indices. Here the notation is simplified as* $\gamma_m \equiv \gamma_{\Theta_m}$.

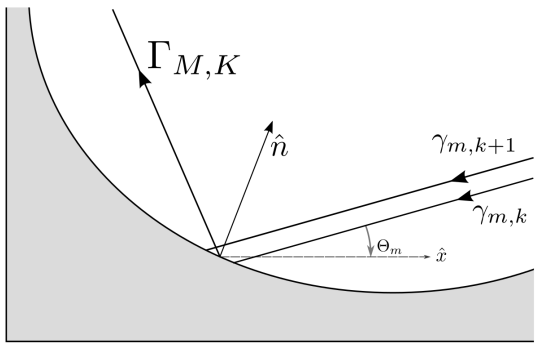

Figure 4.11: *In general the spatial coordinates of the incoming trajectories* γ *will not correspond to that of the outgoing* Γ*, so a linear interpolation between the two nearest trajectories is made. Here the notation is simplified as* $\gamma_{m,k} \equiv \gamma_{\Theta_m, x_k}$.

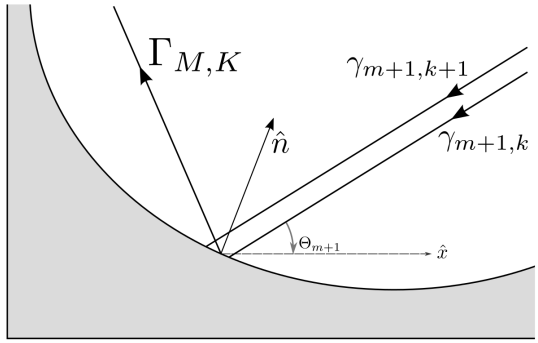

Figure 4.12: *Analogous to figure 4.11, but for* Θ_{m+1}.

provided building blocks, a code library from which a programmer can construct his or her own application. The application in this case being a fully defined superconducting system. Depending on which API components are used and parameters set, different physical systems can be constructed. This means there are no predefined solvers for any complete systems in the API, it all depends on how the components are put together. In a well defined API, the user is isolated from the implementation details by one or more layers of abstractions. This means that the implementation details should be able to change without affecting the behavior of the built application. The API for this thesis project has been developed with this property in mind. For example, the module (class) which solves the Riccati equations does not expose any implementation details. As a user, one only needs to call the compute function and provide some parameters. How the equations are solved internally can be completely arbitrary, as long as it provides the output provided by the API "contract". With this structure, one could implement another method of solving the Riccati equations without affecting anything outside of the class. Following are descriptions of some of the main features implemented in the API.

4.3.1 Physical configurations

Order parameter symmetry: s- and d-wave pairing symmetries are implemented. By default the d-wave is of $d_{x^2-y^2}$ symmetry. By rotating the lattice axis $\pi/4$ to the spatial coordinate system, d_{xy} symmetry is obtained. Any combination of these symmetries can be constructed, such as $d_{x^2-y^2}+\mathrm{i}s$ and $d_{x^2-y^2}+\mathrm{i}d_{xy}$, where each component has its own critical temperature. The d-wave component supports a constant or spatially varying lattice orientation, so that single- or multi-grain systems can be constructed. Only singlet pairings are implemented so far, i.e. p-wave cannot yet be computed for.

Simulation domain and boundary condition: Finite size systems (grains) of arbitrary shape can be computed. Discs and polygonal objects can be used as geometric primitives to shape the grain. The primitives can either be used to add an area, or to subtract from the existing. For example, by adding a disc primitive, and subsequently removing a smaller disc a circular annulus can be defined. Polygons can be removed from discs, and vice versa. The implemented boundary condition is specular reflection, which translates to a perfectly clean superconducting sample in vacuum.

Global physical parameters: The implemented physical parameters affecting the whole system are size, temperature, and external magnetic field (along z-axis). The magnetic field can either be constant or spatially varying along the xy-plane, and enters the equations through a shift in energy by the vector potential.

4.3.2 Computed properties

In this section the different physical properties that the API can currently compute are listed, together with the corresponding expressions for them, and in which units the result is given.

The **order parameter** returned is computed according to

$$\frac{\Delta(\boldsymbol{R})}{k_B T_c} = \lambda \frac{T}{T_c} \sum_{\varepsilon_n \leq \varepsilon_c} \langle \mathcal{Y}^*(\hat{\boldsymbol{p}}_F)(f + (\tilde{f})^*) \rangle_{\hat{\boldsymbol{p}}_F} \tag{4.7}$$

where $f,\tilde{f} = f,\tilde{f}(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\varepsilon_n)$, λ is the coupling constant as defined in (2.4), and $\mathcal{Y}$ is the superconducting symmetry basis function (2.5).

The total quasiparticle **current** is given as

$$\frac{\boldsymbol{j}(\boldsymbol{R})}{2\pi e v_F N_F k_B T_c} = -\frac{T}{T_c} \cdot 2 \sum_{\varepsilon_n \leq \varepsilon_c} \langle \hat{\boldsymbol{v}}_F \cdot g(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\varepsilon_n) \rangle_{\hat{\boldsymbol{p}}_F} \tag{4.8}$$

The **free energy** is

$$\frac{\delta\Omega}{N_F k_B T_c} = \int d\boldsymbol{R} \left\{ |\Delta(\boldsymbol{R})|^2 \ln\frac{T}{T_c} + 2\pi \frac{T}{T_c} \sum_{\varepsilon_n < \varepsilon_c} \left[\frac{|\Delta(\boldsymbol{R})|^2}{\varepsilon_n} - \langle \mathcal{I}(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\varepsilon_n) \rangle_{\hat{\boldsymbol{p}}_F} \right] \right\} \tag{4.9}$$

where

$$\mathcal{I} = \frac{\Delta^* f + \Delta \tilde{f}}{1 + ig} \tag{4.10}$$

As a side note, the formulation in (4.9) is not suitable to implement in the integration module. However, the expression can be rearranged to the following form

to suit

$$\int |\Delta(\boldsymbol{R})|^2 d\boldsymbol{R} \left(\ln\frac{T}{T_c} + 2\pi\frac{T}{T_c}\sum_n \frac{1}{\varepsilon_n} \right) - 2\pi\frac{T}{T_c}\int d\boldsymbol{R} \sum_{\varepsilon_n<\varepsilon_c} \langle \mathcal{I}(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\varepsilon_n)\rangle_{\hat{\boldsymbol{p}}_F} \tag{4.11}$$

This way, the first term can easily be computed, given Δ, while the second term with the momentum integral can be computed with the integration module (see section 4.4 regarding the ComputeProperty and IntegrationIterator classes).

The **local density of states** is given per unit energy as

$$\frac{N(\boldsymbol{R},\epsilon)}{N_F} = -\mathrm{Im}\langle g(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\epsilon)\rangle_{\hat{\boldsymbol{p}}_F}, \qquad \epsilon = \varepsilon + i0^+ \tag{4.12}$$

And finally, the **spectral current density** is given as

$$\frac{\boldsymbol{j}(\boldsymbol{R},\epsilon)}{2\pi e v_F N_F} = -\frac{1}{2\pi}\mathrm{Im}\langle \hat{v}_F \cdot g(\boldsymbol{R},\hat{\boldsymbol{p}}_F;\epsilon)\rangle_{\hat{\boldsymbol{p}}_F}, \qquad \epsilon = \varepsilon + i0^+ \tag{4.13}$$

4.4 API class descriptions

Now that some broad information about the framework has been given, it is suitable to go into more detail and present the different classes of the API, and how they work together. The description will still be relatively brief and explanatory. For a more detailed description, the reader is referred to the code comments in the header files.

The code library can roughly be divided into three kinds of classes; internal API classes, external API classes, and utility classes. The internal classes are created and used by the API, but are not exposed to the user. External API classes are the ones instantiated by the user to set up a system. Utility classes are used by the user and/or framework, but are not strictly tied to it. Put differently, these classes may also be of use outside of this particular framework. The external API classes generally deals with defining system parameters, order parameter symmetry, grain geometry, type of solver for the Riccati equations, and telling the solver what type of physical properties we want to be computed. These objects are passed on to the internal API classes, where a ”machinery” exists to solve the equations based upon said input. Central to the internal API is the class which ties the different parts together, and orders the computation of the momentum integral and the requested physical properties which follows from it. But it is perhaps best to begin by introducing the most widely used class, a utility class used to store and manage data.

In the following text, all names which refer to an API class, a function call, or other fractions of actual code are printed in `verbatim` as per convention. Also,

when specifying a template parameter in the angle brackets (section 3.1.1) in the following text, the placeholder `T` is often used (not be be confused with temperature) instead of an actual data type. For example `Context<T>` instead of `Context<double>`. This is per convention in C++, and can be turned into valid code by using the following line at the beginning of the code listing:

```
typedef double T;
```

which will replace all instances of `T` with `double` at compile time.

4.4.1 GridData

The `GridData` class is written to eliminate the cumbersome work associated with resource management, and also to provide basic data manipulation functionality. All data allocated using native code is returned as a 1-dimensional array, which is how it is laid out in memory. If a more complex data structure is to be used, say a complex valued 2-dimensional field, the data has to be logically partitioned according to a certain pattern. Also, the partitioning must be carefully chosen such that we get optimal performance when operating on the data. Finding the right address in memory to the data we want to access can be a very error prone operation, if done manually. The `GridData` class provides a convenient handle to its underlying data. It can handle 1- or 2-dimensional data, real or complex valued, and vector fields of these with any number of vector components. The data is also partitioned in such a way as to be optimal for use with a GPU. The partitioning can be seen in Figures 4.13, 4.14. A `GridData` object can allocate memory on either the CPU or the GPU. Convenience functions are written to return a memory pointer to the sought location (CUDA kernels are called with native data pointers). Most of the arithmetic operators are implemented, and a number of other operations can be applied, such as complex- and vector field magnitude, rotation, and copying of individual fields. Moreover, it provides a convenient way to copy data between the CPU and the GPU. File operations are also implemented so that data can easily be stored and retrieved. `GridData` is a template class, which in this case means it can be instantiated for any basic data type, i.e. we can store arrays of `float`, `double`, `int`, etc. Much of the core functionality is built upon the thrust library [38], included in the CUDA SDK (Software Developer Kit). Any operations expressed in terms of thrust function calls automatically works for data on the CPU or GPU. Otherwise two separate functions have to be written, if global functionality is desired.

When a large array of data is passed around in the API, it is nearly always done so with `GridData` objects. To demonstrate some functionality, the code below will instantiate a complex valued 2-dimensional data array on the GPU with $(N_x,N_y) = (100,100)$ data elements of type `double` (double precision floating point), followed by some simple operations.

(Note that `GridData` objects will actually be instantiated with the naming `Grid`.

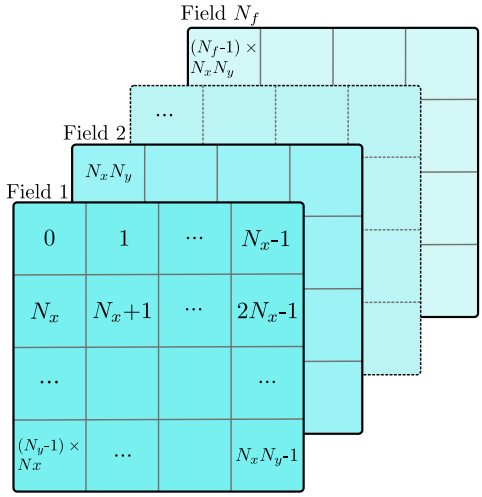

Figure 4.13: *Logical memory partitioning of internal data to the* ***GridData*** *class. As is conventional in C, data is stored in row-major, or x-axis-major, order.*

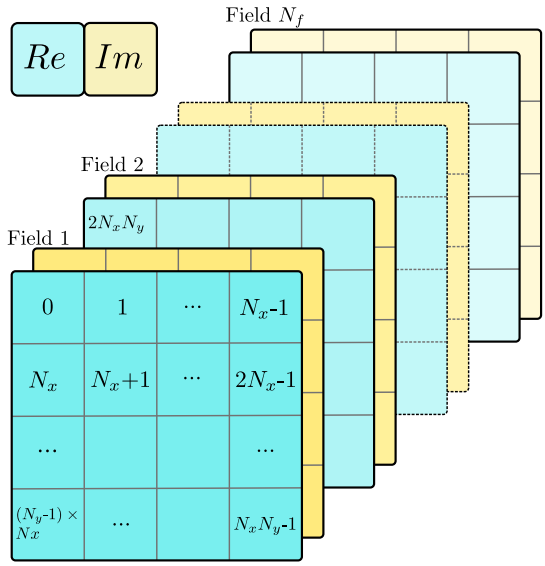

Figure 4.14: *Logical memory partitioning of complex valued data in the* ***GridData*** *class. By keeping the real part and the imaginary part separate (and not interleaved), coalesced memory is achieved when accessing data from a GPU threadblock.*

This is a semantic redefinition to simplify syntax with template usage. The class, however, is in fact called `GridData` in its definition. The type `Grid<double>::GPU` as written in the code below is identically the same as `GridData<double,thrust::device_vector<double>>`. We believe most people would prefer the first syntax.)

```
// Instantiate object and allocate GPU memory (1 field component)
Grid<double>::GPU myGrid(100,100,1,Type::complex);

// Get data pointers to real and imaginary parts
// 0 specifies field component index
double* re = myGrid.getDataPointer(0,Type::real);
double* im = myGrid.getDataPointer(0,Type::imag);

// Call a CUDA kernel and do some calculations
some_kernel<<<blocksPerGrid,threadsPerBlock>>>( re, im );

// Copy data to CPU
Grid<double>::CPU myGrid_cpu = myGrid;

// Write data to disk
myGrid.write("path/to/file.grd");
```

There are no convenience functions/operators implemented to modify individual data elements, as accessing data on the GPU this way would be extremely inefficient.

4.4.2 Context and ContextModule

The API consists of a number of different classes, each representing a logical delimited part of the whole system. However, each component usually require access to the data hosted by one or more of the other classes. One way to pass data is by function argument, an approach which can be quite tedious if there is a significant amount of communication taking place. Also, in the early stages of development when the structure changes frequently, the functions may have to be constantly rewritten to reflect the changes. A convenient workaround is to introduce a class which knows about the existence and whereabouts of the major components which makes up the system. In OOP such a class/object is often called a *context*. Moreover, each of the class members associated with this context also knows its location. In our API, the context class is simply called `Context`. All classes associated with the context are derived from a base class called `ContextModule`. This way the code dealing with the association only have to be written once. Now, any class derived from `ContextModule` has access to all other such classes, as long as they are associated with the `Context` class, without having to pass any function arguments. The drawback with this approach is that

it is harder to determine the external dependencies, as one has to look at the actual function code rather than just the function arguments.

Furthermore, the `ContextModule` base class also has a *pure virtual* function called `initialize()` which should (re)allocate all data needed by this class. A pure virtual function means it must be implemented or the compilation will fail. In conjunction, the `Context` class also has an `initialize()` function, which in turn calls the corresponding `initialize()` function of all its associated `ContextModules`. This should be done before starting a computation, or when some parameter has changed. In general, what normally happens in the constructor has been deferred to the `initialize()` function. The reason being that there are some dependencies between the classes, and by letting the context do the constructor work (initializing), it can do things in the correct order without the user having to worry about it. The `Context` class also contains a number of convenience functions to easier deal with central and commonly used operations. The following code demonstrates how the association between `Context` and `ContextModule` is made, i.e. by constructor argument:

```
// Create a new Context
Context<T>* ctx = new Context<T>;

// ctx and param will now be associated
Parameters<T>* param = new Parameters<T>( ctx );

// ctx and delta will now be associated
OrderParameter<T>* delta = new OrderParameter<T>( ctx );

// ...create more API components

// Initialize all Context members and compute one
// self consistent iteration
ctx->initialize();
ctx->compute();
```

The following classes are `ContextModules`: `Parameters`, `OrderParameter`, `GeometryGroup`, `RiccatiSolver`, `IntegrationIterator`, `ComputeProperty`, and `GradientAccelerator`. Each of these, along with some internal API classes, will be described below in some detail.

4.4.3 Parameters

The `Parameters` class is simply a structure to collect all parameters that affect the system *globally*. There are many system parameters which reside elsewhere, usually in the corresponding class which it affects, but only when it does not affect anything directly outside of the class. Examples of global parameters are temperature, physical size, and external magnetic field, and also numeric parameters

such as discrete resolution in real and momentum space. An example of a local parameter which is **not** held in the `Parameters` class is the shape definition of the finite grain geometry. This of course affects the solution, but no other class needs to know about these parameters, so they are kept locally in the geometry class.

4.4.4 OrderParameter and OrderParameterComponent

The `OrderParameter` class, perhaps unsurprisingly, holds the details regarding the superconducting order parameter Δ. It contains the current estimate of the order parameter field $\Delta(\boldsymbol{R})$ (or more precisely $\Delta(\boldsymbol{R},\boldsymbol{p})$), and also what type of order parameter symmetry, or combinations of symmetry, are present in the system. Currently, any combinations of s- and d-wave symmetries are possible. The `OrderParameter` class is a container class with some utility functions. The contained classes, which ultimately defines the order parameter, are of the type `OrderParameterComponent`. This class is an abstract base class (meaning it has pure virtual member functions), from which the particular order parameter symmetry classes have to be derived and implemented. How to create an s-wave order parameter symmetry is illustrated below:

```
// Create container class (assuming Context ctx has been created)
OrderParameter<T>* delta = new OrderParameter<T>( ctx );

// Create s-wave symmetry
OrderParameterComponent_S<T>* s = new OrderParameterComponent_S<T>;

// Set initial condition as bulk value, using Initial utility class
Initial<T>::bulk( s );

// Add symmetry component to order parameter container
delta->add( s );
```

If d_{xy}+is order parameter symmetry is desired, it is a simple matter of adding a d-wave component, setting initial guess and optional parameters

```
// Create d-wave symmetry
OrderParameterComponent_D<T>* d = new OrderParameterComponent_D<T>;

// d-wave symmetry depends on atomic lattice orientation, which
// can be set arbitrarily (relative to x-axis). Default is zero.
d->setLatticeOrientation( PI/4.0 );

// Set initial condition as bulk value
Initial<T>::bulk( d );
```

```
// Add symmetry component to order parameter container
delta->add( d );
```

And, as a final example, to create a system with $d_{x^2-y^2}+\mathrm{i}d_{xy}$ order parameter symmetry, create the following two `OrderParameterComponent`s

```
OrderParameterComponent_D<T>* d1 = new OrderParameterComponent_D<T>;
OrderParameterComponent_D<T>* d2 = new OrderParameterComponent_D<T>;

d1->setLatticeOrientation( PI/4.0 ); // d_xy
d2->setLatticeOrientation( 0.0 );    // d_x2-y2

Initial<T>::bulk( d1 );
Initial<T>::constant( d2, 0.0 );

delta->add( d1 );
delta->add( d2 );
```

Note that while `OrderParameter` is a `ContextModule`, `OrderParameterComponent` is not.

4.4.5 GeometryGroup and GeometryComponent

The description of the physical boundary of the system (herefter referred to as the geometry), and its discrete representation, is managed by the `GeometryGroup` class. Structurally, the `GeometryGroup` and `GeometryComponent` classes are very similar to the `OrderParameter` and `OrderParameterComponent` classes. The `GeometryGroup` is a container class for `GeometryComponent` objects, with some utility functions. The geometry of the physical system can currently be defined in terms of discs and polygons. The corresponding classes are `DiscGeometry` and `PolygonGeometry`, which both derive from `GeometryComponent`. Any number of components can be added, but for reasons of efficiency, the number should be kept as low as possible. Any component can either be set to logically *add* or *remove* to the geometry. So, for example, a square with a circular hole inside can be created by first adding a `PolygonGeometry` with 4 vertices (specifying the corners) set to logical add, and subsequently adding a `DiscGeometry` set to logical remove. They also must be added in this order. The corresponding code would look like:

```
// Create geometry container
GeometryGroup<T>* geom = new GeometryGroup<T>( ctx );

geom->add( new PolygonGeometry<T>( 4, outerRadius ), Type::add );
geom->add( new DiscGeometry<T>( innerRadius ), Type::remove );
```

It should be pointed out here that the `PolygonGeometry` class approximates a circle with N vertices by default. A simple way to create a square is by approx-

imating a circle with 4 vertices. One can of course also create a general polygon shape by specifying explicit vertex coordinates to the constructor.

So far only a formal description of the geometry has been defined. When solving the Riccati equations a precomputed discrete representation of this geometry is used, for reasons of efficiency. This representation is comprised of a discrete lattice of geometry *labels*, another lattice where the exact distance to the boundary is stored, and finally a third vector field lattice which contains the exact normal at the exact location of the boundary. The label lattice defines whether a specific lattice point is inside the geometry, outside of the geometry, or at/near the boundary. With this representation, the Riccati equations can be solved exactly up to the boundary coordinate without having to approximate the geometry to the lattice resolution. The normal vector field is used to apply the specular boundary condition. To generate the discrete representation for any rotational frame, the `GeometryGroup::create()` function should be called with the desired rotation in radians as input argument. The resulting fields are stored as class member variables, and can be retrieved by using the appropriate `get()` function.

4.4.6 RiccatiSolver

As the name implies, the `RiccatiSolver` class is responsible for solving the Riccati equations. However, this is just a base class where some of the functions are marked as pure virtual. Thus, this class only provides some basic functionality and defines the interface of how a Riccati solver communicates with the rest of the API. The motivation is that while the currently implemented solver employs the method described in section 2.1, there should not be anything ruling out future solvers to be implemented with a different technique for solving differential equations, like Runge-Kutta. As long as the solver adheres to the interface provided by the `RiccatiSolver` base class, it can be implemented in any way desirable. There may also be different solvers optimized for different systems, like bulk systems or finite systems, clean or with impurities, singlet or triplet pairing. Currently, only one solver is implemented, `RiccatiSolverConfined`, which is capable of handling clean, finite sized systems with singlet pairing mechanisms (i.e. scalar coherence functions).

The syntax for creating an instance of `RiccatiSolverConfined` looks like

```
RiccatiSolver<T>* solver = new RiccatiSolverConfined
                <T, Gamma::General, VectorPotential::Radial>( ctx );
```

A few things to note here. The first template argument is just the datatype we want to store the data in (i.e. `float` or `double`). The second template argument specifies which function to use when integrating the coherence functions. There are different functions optimized for different circumstances, but `Gamma::General` will work for all cases. The third template argument defines which gauge field $\boldsymbol{A}$ to use when incorporating an external magnetic field. Here we have used a radially symmetric form, which is defined as $\boldsymbol{A} = \frac{B_0}{2}(-y\hat{x} + x\hat{y})$, which corresponds to

the magnetic field $\boldsymbol{B} = B_0\hat{z}$. If one does not wish for an external magnetic field, `VectorPotential::None` should be used.

By calling the member function `computeCoherenceFunctions()`, the Riccati equations should be solved for both coherence functions in real space, for all energies, for a given rotational frame. Thus, when the function returns, we should have $\gamma,\tilde{\gamma}(\boldsymbol{R},\epsilon)|_{\hat{\boldsymbol{p}}_F}$. The result is currently stored in member variables (of type `GridData`), and can be accessed by retrieving pointers through the `getGamma()` and `getGammabar()` member functions.

The `RiccatiSolver` class is closely linked with the functionality controlling the boundary condition(s), i.e. the `BoundaryStorage` and `BoundaryCondition` classes.

4.4.7 BoundaryStorage

Due to the nature of the implemented Riccati solver, and to the requirements of any finite size boundary condition, there is a need to record and store the values of the coherence functions at the boundary. The incoming trajectory (towards the boundary), integrated along momentum $\boldsymbol{p}_i$, will reflect specularly against the boundary normal and acquire momentum $\boldsymbol{p}_o$. But since we only integrate along one momentum direction at a time, this value (at the boundary) will have to be stored so we can resume the integration at a later time. This means that the boundary values of γ and $\tilde{\gamma}$ will have to be stored for all energies/Matsubara frequencies and all discrete momenta. One quickly realizes that all these values will have to be stored in some kind of sparse storage, otherwise we will be limited to very low discrete resolutions. The `BoundaryStorage` class handles the sparse storage and retrieval of boundary values for the coherence functions. The structure of the sparse storage is determined by analyzing the discrete representation of the geometry for all discrete momenta. Various helper arrays with indices and pointer offsets are constructed to find the right location in the sparse storage for any given trajectory. What this class does not do is to actually read and write values to these arrays, it only provides the means to do so. The values are read and written into these arrays by the Riccati solver, and as such the solver kernel must be given the right index helper arrays from this class. Moreover, once all the boundary values have been stored, there must be a function to reflect the incoming trajectories to the corresponding outgoing ones, based on the boundary normal. Effectively a complete reshuffling of all the elements in the sparse array, a task not at all trivial, even for a simple operation like a specular reflection. This operation is handled by the `BoundaryCondition` class.

4.4.8 BoundaryCondition

Being a member of the `RiccatiSolver` base class, the `BoundaryCondition` class is itself an abstract base class. It has a function called `computeBoundaryCondition()` which is marked as pure virtual, meaning it must be implemented in any derived

class. The only current implementation is the `BoundaryConditionSpecular` class. The `RiccatiSolver` class can host any number of `BoundaryCondition` objects, implying that multiple boundary conditions for any given system is possible. However, this is not implemented yet. A boundary condition is specified according to the following code example

```
RiccatiSolver<T>* solver = new RiccatiSolverConfined
                 <T, Gamma::General, VectorPotential::None>( ctx );

solver->add( new BoundaryConditionSpecular<T> );
```

To add another boundary condition, additional code could look like

```
solver->add( new BoundaryConditionBulk<T> ); // Not implemented yet
```

Although, if multiple boundary conditions are implemented, one must find a way to specify unique coordinates for each condition, in order to avoid multiple conflicting conditions at the same coordinate.

4.4.9 ComputeProperty

Just like with an experiment, when running a simulation there are certain physical properties of interest, and it is not always we need to know about everything. Computing each physical property (superconducting order parameter, current, free energy, etc) takes time, and by only computing the desired property, or properties, the simulation will run faster. For this purpose a modular setup has been employed, where each physical property to be computed is represented by a class. All these classes must derive from the `ComputeProperty` base class, which has a number of pure virtual functions which need to be declared. These functions, in turn, connect to how the momentum integral is executed in the `IntegrationIterator` base class. The obligation of the `ComputeProperty` class is to compute the integrand of the momentum integral only, while the actual integration is taken care of by said `IntegrationIterator` class. This way code duplication is reduced, and multiple physical properties can be computed efficiently.

To request a particular property to be computed, the corresponding class object is instantiated and given to the context:

```
new ComputeOrderParameter<T>( ctx );
new ComputeCurrent<T>( ctx );
```

After the contexts `compute()` function have been called, the result can be retrieved from the compute objects by calling `getResult()`. But first we must find the compute object itself by asking the context:

```
ctx->compute();
```

```
ComputeProperty<T>* j_obj = ctx->getComputeCurrent();
Grid<T>::GPU* j_data = j_obj->getResult();

// Or chain the calls for less code
// Grid<T>::GPU* j_data = ctx->getComputeCurrent()->getResult();
```

We can now choose to write the result to disk or to plot it to the screen, or both. The result can be written to disk in one line:

```
ctx->getComputeCurrent()->getResult()->write("filename.grd");
```

The classes derived from `ComputeProperty` are: `ComputeOrderParameter`, `ComputeCurrent`, `ComputeFreeEnergy`, `ComputeLDOS`, `ComputeCurrentDensity`.

4.4.10 IntegrationIterator

The `IntegrationIterator` is categorized as an internal API class, meaning the user never explicitly have to create, or communicate with the class. It is, however, a central part of the system, as it is responsible for coordinating the momentum integral $\int(...)d\hat{\boldsymbol{p}}_F$, where it for each step communicates with the appropriate objects to generate required data and direct it to the intended receiver. Conceptually, the integration loop has the following form (with the classes responsible for the operation in parenthesis):

For each discrete momentum direction in the momentum integral:

- Compute discrete geometry representation (`GeometryGroup`)
- Retrieve order parameter in current rotational frame (`OrderParameter`)
- Solve Riccati equations for current rotational frame, for all energies (`RiccatiSolver`)
- Compute integrand of requested physical properties, for all energies (`ComputeProperty`)
- Sum over energies (if Matsubara) (`ComputeProperty`)
- Add current momentum contribution to full integral (`ComputeProperty`)

The boundary condition is applied before the momentum integral. For spectral properties like LDOS and current density, the energy sum is not performed.

4.5 Usage

With the framework described, it is apt to present and describe a program which sets up a simple system and runs the self-consistent iterations. The following code will set up a system identical to the one treated in paper I.

First we define the data type to be double precision. By using `typedef` we can simply write `T` instead of `double` at each occasion. It is also easy to change to the faster but less accurate `float` data type (for testing purposes) by just changing the first line.

```
typedef double T;
```

Define some constants which we use later when creating the geometry.

```
const T outerRadius = 0.48;
const T innerRadius = outerRadius * 0.35;
```

Next we create the first API object, which is the context. This object will keep track of the various other API components. It also has a number of convenience functions.

```
Context<T>* ctx = new Context<T>;
```

Create a parameter object and set some values like discrete resolution and temperature.

```
Parameters<T>* param = new Parameters<T>( ctx );

param->setGridResolution( 400 );
param->setGridWidth( 10.0 );
param->setAngularResolution( 200 );
param->setTemperature( 0.1 );
param->setConvergenceCriterion( 1.0E-6 );
```

The following lines first creates an empty geometry container. The actual shape descriptions are added to it afterwards. The first geometry shape is a square. Next we define a smaller square which will cut a hole in the one first added. The last argument defines if the geometry should add or remove to the geometry description.

```
GeometryGroup<T>* geom = new GeometryGroup<T>( ctx );

geom->add( new PolygonGeometry<T>( 4, outerRadius, 0.0 ), Type::add );
geom->add( new PolygonGeometry<T>( 4, innerRadius, 0.0 ), Type::remove );
```

Add a solver for the Riccati equations. The derived class `RiccatiSolverConfined` can handle the specular boundary condition. We want it to handle the general form of γ, and we use an empty function for the vector potential, since there is no external magnetic field.

```
RiccatiSolver<T>* solver = new RiccatiSolverConfined
                          <T, Gamma::General, VectorPotential::None>( ctx );
```

Add a boundary condition. Currently, only specular boundary condition is implemented.

```
solver->add( new BoundaryConditionSpecular<T> );
```

Create empty order parameter container, and add d-wave symmetry component.

```
OrderParameter<T>* delta = new OrderParameter<T>( ctx );

OrderParameterComponent_D<T>* d_wave =
                                    new OrderParameterComponent_D<T>( param );
delta->add( d_wave );
```

Set initial condition and atomic lattice orientation for the d-wave component. `Initial<T>::random()` sets the order parameter to its constant bulk value, but adds a spatial stochastic component on top. This helps the order parameter find its true ground state. By setting the lattice orientation to $\pi/4$, we get maximum pair-breaking at the interface.

```
Initial<T>::random( d_wave, 0.01, 0.25*M_PI );
d_wave->setLatticeOrientation( 0.25*M_PI );
```

The last API objects we need to create are the compute objects. These represent the physical quantities we want the solver to compute for us. Here we request the order parameter (since it is a self-consistent iteration), and also the quasiparticle current.

```
new ComputeOrderParameter<T>( ctx );
new ComputeCurrent<T>( ctx );
```

Before we start the compute process, we must initialize all API members correctly. The `Context` class provides a convenience function to do this for us.

```
ctx->initialize();
```

Calling `Context::compute()` performs one self-consistent iteration. We want to loop this until we reach convergence, as specified in the parameter object.

```
while ( ctx->isConverged() == 0 )
{
  ctx->compute();
}
```

Finally, when convergence has been reached, we extract the computed information and save it to disk. By passing the pointer to the parameter object as an argument, it is stored together with the data. This is highly recommended as it can be very hard to keep track of otherwise.

```
Grid<T>::GPU* delta_grid = ctx->getComputeOrderParameter()->getResult();
Grid<T>::GPU* current_grid = ctx->getComputeCurrent()->getResult();

delta_grid->write("delta_filename.grd",param);
current_grid->write("current_filename.grd",param);
```

Once the data has been saved to disk, it can be viewed and analyzed with the accompanying `viewgrid` command line tool. It can also be written in the Numpy array format, so one can use their favorite python tools to further examine or plot the data.

To summarize, in the above code we have defined a complete superconducting system with multiply connected geometry, requested the order parameter and current to be computed, run the self-consistent iterations until convergence, and saved the result to disk, all in about 30 lines of code. Add 3 or 4 more lines for an additional order parameter symmetry, and we could instead compute for a d+is or $d_{x^2-y^2}$+id_{xy} system.

Chapter 5

Results

When implementing any new code, it must be established that the output is correct and the desired one. No matter how prudent one has been when authoring, there is always the (rather large) possibility of error. Therefore it is necessary to somehow verify the output. Here, this is achieved by *unit testing*, where sections of code are tested in isolation where possible, and also by attempting to reproduce a number of well known physical phenomena which can be compared to measured results and theory.

One of the benefits of using an existing software over writing new code is that the former has already been verified. In this chapter, benchmark and validation results are presented as evidence of the reliability of the developed software, and some newly discovered results are discussed.

5.1 Scaling

Of interest in a computational context is how well the problem scales in the implementation, i.e. how it performs under change in problem size and number of processors put to work. Arguably, the most common measure for a parallel solver is to see how much faster the problem is computed, depending on how many processors, or cores, are employed. In the case of a GPU however, we must always use all cores, so this measure is not available on a single unit. What we can measure instead is the computational speed as a function of problem size. This measure gives us some hint on how efficiently the GPU is put to use. Plotted in Figure 5.1 is the time taken per computational element, as a function of spatial resolution and Matsubara frequencies. Both these dimensions are trivial to compute in parallel (an *embarrassingly* parallel problem, as per computer science terminology) in our implementation. At the lowest discrete resolution we can see that it takes about 4 times longer to compute one lattice element, compared to the fastest setup. The efficiency is low here because there is not enough parallel

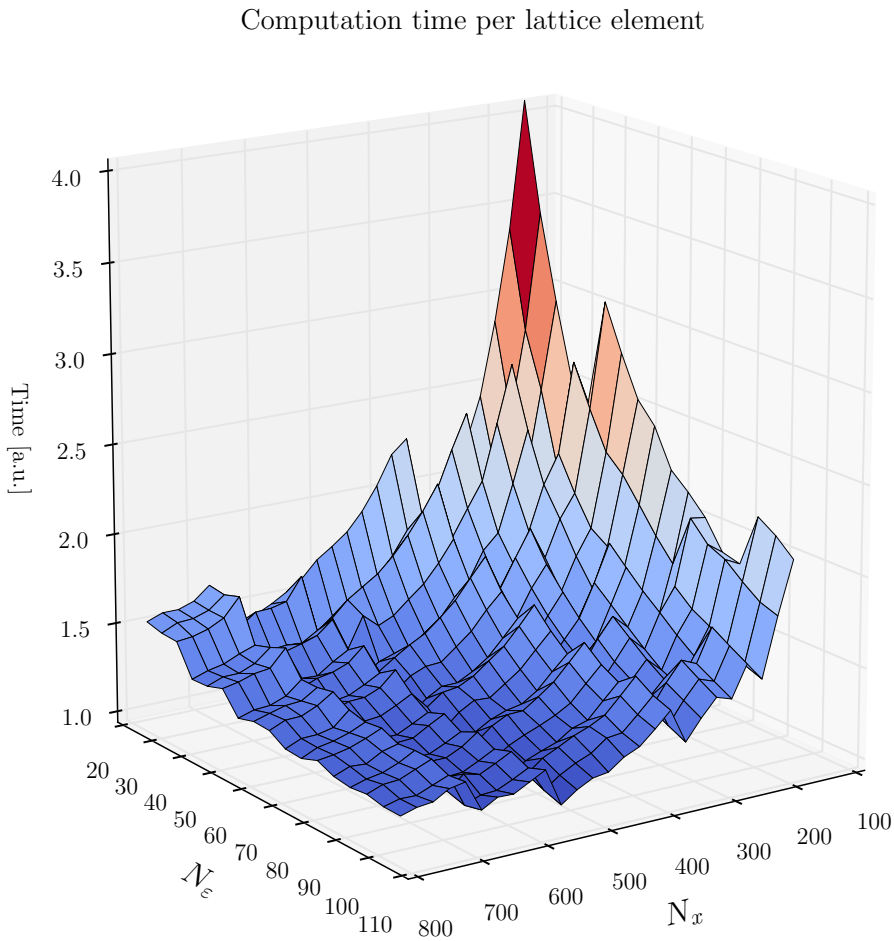

Figure 5.1: *Mapping of time per computational element for different problem sizes. N_x is the number of elements along x, and N_ε the number of Matsubara frequencies. The total number of elements is thus $N = N_x^2 N_\varepsilon$. The time has been normalized to the most efficient configuration. For small lattice sizes there is not enough parallelism in the problem to utilize all the cores. For higher spatial resolution or increasing Matsubara frequencies we soon saturate the GPU with work and the efficiency increases, resulting in a much lower time per computational element.*

work to keep all the cores busy. Furthermore, for a small lattice resolution, if the spatial lattice is not evenly divisible by 32, the amount of 'lost' cores will be large in relation to the total amount of cores. The silver lining here is that the problem quickly becomes more efficient as we increase either the number of Matsubara frequencies, or increase the spatial resolution. At $N_x = 400$ we are very close to optimal configuration.

5.2 Convergence

Depending on the system being solved for, the time to convergence (number of self-consistent iterations) can vary considerably. If the initial condition is set close to the actual solution, the process is usually relatively fast; a satisfactory solution can be had in less than 100 iterations, which translates to less than 10 minutes for a system at $T = 0.1T_c$ ($N_\varepsilon \approx 200$) at $N_x = N_y = 400$ lattice elements, and discretizing the momentum integral to $N_{\theta_{p_F}} = 100$ angles. In cases where we don't know an approximate solution, and/or when the free energy landscape is flat, like the case described in paper I, the number of required iterations can increase greatly. To reduce the time to reach convergence, a simple "accelerator"has been implemented. Figure 5.2 shows a plot of the free energy and residual measure as a function of iteration count. The accelerator works on the simple principle of guessing the 'direction' of the solution (Δ) based on the last two iterations, followed by applying a factored version of that direction to the solution

$$\Delta_{i+1} \longleftarrow \Delta_{i+1} + \mathcal{A}(\Delta_i - \Delta_{i-1}), \quad \mathcal{A} \sim \text{Min}\left[a, \frac{1}{\text{residual}}\right], \tag{5.1}$$

where the factor $\mathcal{A} > 1$ is determined by the current residual, and a is a safeguard to avoid applying too much of the guess. The solution is allowed to relax for a few iterations before applying another guess. The resulting gain varies between cases, but the accelerator typically provides a speedup of a factor 2 to 5.

5.3 DOS profiles

The density of states (DOS) around the energy gap has a distinct profile for superconductors with different pairing symmetries, and is easily recognizable for an ideal clean system. The superconducting energy gap is also a well known value. Moreover, when measuring, the DOS is an important signature in identifying the type of pairing symmetry in a superconductor. Many of the unconventional superconductors are believed to have a combination of pairing symmetries. For example, YBCO is believed to predominately have a $d_{x^2-y^2}$ symmetry, but there is evidence of a subdominant s-wave pairing. The emergence of a subdominant order parameter will leave a trace in the DOS profile, as we will demonstrate in the next section. Presented in Figure 5.3 are the DOS spectra for s- and d-wave pairing symmetries.

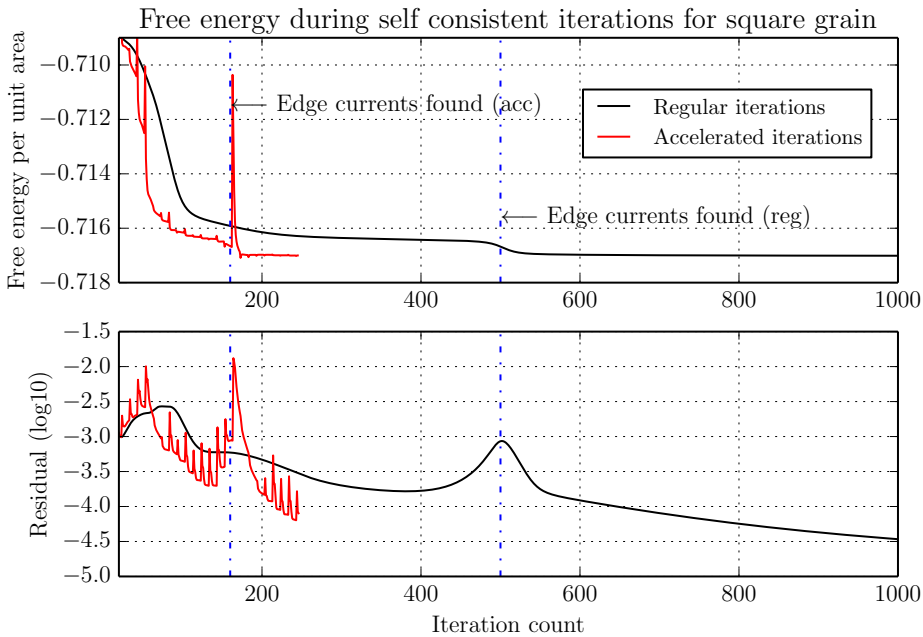

Figure 5.2: *Free energy and residual as a function of self-consistent iteration count, comparing regular unaltered iterations and with applied convergence acceleration. The system is identical to that in paper I, i.e. a square d-wave grain with pair-breaking edges, at $T = 0.1T_c$. We see that by applying acceleration, the convergence rate is significantly sped up. There is a spike in the curve every time the acceleration is applied, since it imparts a relatively big change in the solution.*
It is also interesting to note that up until the 500th iteration (in the regular case) the edge currents have not been developed yet (solver has not found the ground state), but quite abruptly the local minimum is escaped and the edge current state is found, accompanied by a lower free energy. In this simulation the first guess of the solution $\Delta = \Delta_0$ is a complex valued constant. If the initial guess is instead perturbed by some noise $\Delta = \Delta_0 + \delta(\boldsymbol{R})$, where $\mathrm{Max}[\delta] \approx \Delta_{d,bulk} \cdot 10^{-2}$, the edge current phase is found much faster. On the other hand, if the initial guess is purely real, i.e. $\mathrm{Im}[\Delta] = 0$, the solver will never find the edge currents.

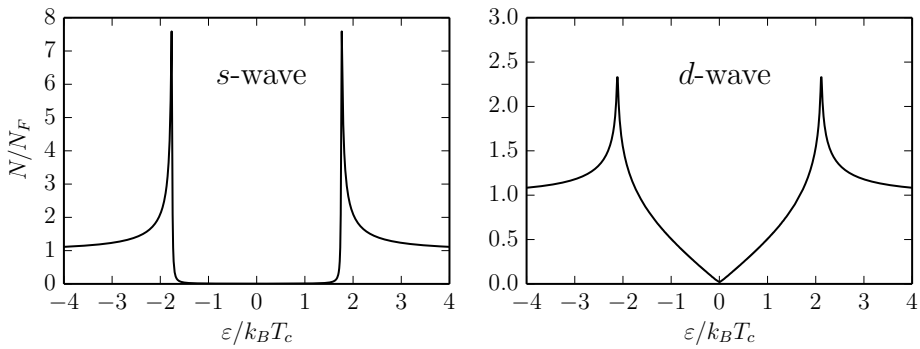

Figure 5.3: *Density of states (DOS) spectrum for a superconducting square grain at $T = 0.3T_c$ with s-wave (left) and d-wave (right) pairing symmetry. For the d-wave case, the atomic lattice is aligned with the edges of the grain, producing no pair-breaking by the grain edges. The edge of the energy gap lies at $1.76k_BT_c$ for the s-wave, and the DOS peak for the d-wave is at $2.12k_BT_c$.*

5.3.1 Midgap states

The DOS profile plotted in Figure 5.3 (right) belongs to the bulk interior in a d-wave superconductor, or a finite size square grain with all sides aligned with the crystal axes. However, when the sides of a grain does not line up with the atomic lattice in a d-wave superconductor, the order parameter gets suppressed (pair-breaking) along the edges [39, 40]. We get maximum pair-breaking when the side is at $\pi/4$ angle to the crystal axes. Figure 5.4 illustrates both these cases. The pair-breaking behavior is a consequence of the sign change in the $\boldsymbol{k}$-dependent order parameter upon $\pi/2$ rotation. The normal region formed along the edges effectively creates a NS interface, or rather a vacuum-normal-superconducting interface. In such regions a process known as Andreev reflection can take place. An electron in the normal phase with energy $|E| < \Delta$ cannot excite any states within the superconducting condensate. What happens instead is that the electron becomes retroreflected as a hole, allowing for a Cooper pair to form in the condensate. In our case the returning hole reflects specularly on the vacuum-normal interface and again approaches the condensate, only to be reflected again but this time as an electron. This process (Figure 5.5) repeats and is called an Andreev bound state (ABS), and gives rise to a peak in the DOS spectrum at a narrow range around the Fermi energy. These are the *midgap states* (Figure 5.6 (right)), and the peak, measurable by tunneling conductance, is a good indication that a d-wave pairing symmetry exists [41]. However, the midgap states are not energetically favorable, and any process which can shift them away from the Fermi energy will be favored. One such "process" is the emergence of a subdominant superconducting order parameter [14, 42–44], if the material supports it. Plotted

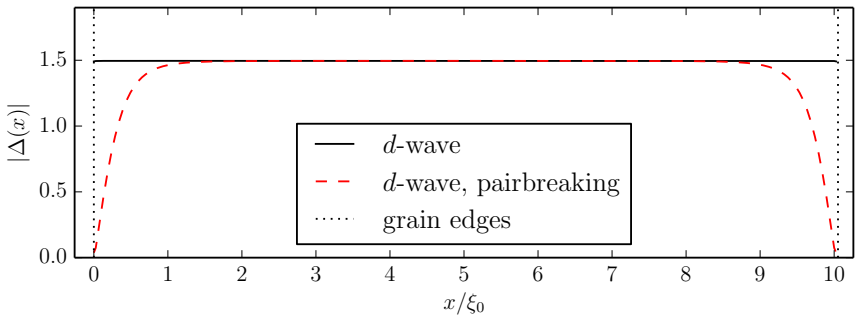

Figure 5.4: *Cross section of the order parameter over a square grain. When the atomic lattice is $\pi/4$ to the grain edge, maximum pair-breaking occurs and the order parameter becomes suppressed (red, dashed). The length scale is $\xi_0 = \hbar v_F / k_B T_c$.*

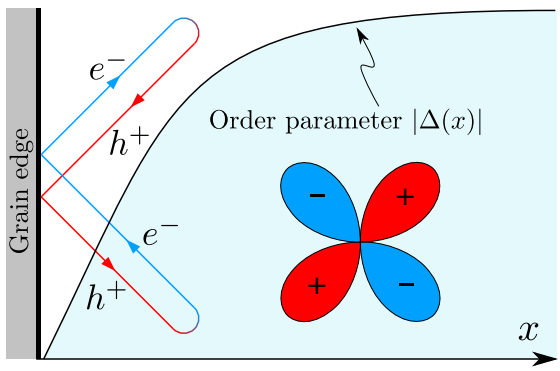

Figure 5.5: *Andreev reflection at the grain boundary gives rise to Andreev surface bound states and a corresponding DOS peak around the zero energy (midgap states).*

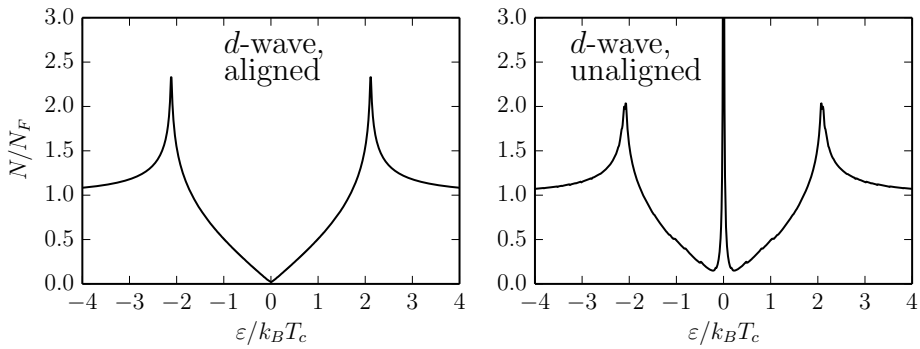

Figure 5.6: *DOS spectrum for a superconducting square grain at $T = 0.3T_c$ with d-wave pairing symmetry. In the left plot, the atomic lattice is aligned with the sides of the grain. In the right plot, however, the atomic lattice is maximally misaligned ($\pi/4$) with respect to the edges of the grain, producing pair-breaking along the grain edges. Here Andreev bound states emerge, resulting in a peak around the zero (Fermi) energy in the DOS spectrum, known as the midgap states.*

in Figure 5.7 is the DOS spectrum for both d+is and $d_{xy} + id_{x^2-y^2}$ pairing in a grain with pair-breaking edges, and it can clearly be seen that the midgap states have been shifted away from the Fermi energy in both cases.

5.4 Free energy

Using the expression (4.9) together with the area of the simulated grain, the free energy per unit area can be computed. This can then be compared with the corresponding analytical expression, an approximation at $T = 0$, where $\delta\Omega = \Delta^2/2$. Since zero temperature is impossible to compute for, we have to settle for a sufficiently low temperature. For an s-wave superconductor at $T = 0.1T_c$, the computed value becomes $\sim 1.51\ k_BT_c/N_F\xi_0^2$, to be compared with $\Delta_s^2/2 \approx 1.55$.

5.5 Abrikosov vortex lattice

By applying an external magnetic field, flux vortices will enter the superconductor as a response. The amount of vortices will correspond to the amount of magnetic flux passing through the superconductor. It is observed that the vortices will arrange themselves in a triangular lattice, a so called vortex lattice, as predicted by Abrikosov [9]. This lattice can be reproduced numerically by choosing an appropriate magnetic field strength for the size of the system, i.e. one that is large enough to fill the superconductor with vortices, but not above H_{c2} as the superconductivity will be destroyed. The result can be seen in Figure 5.8. If

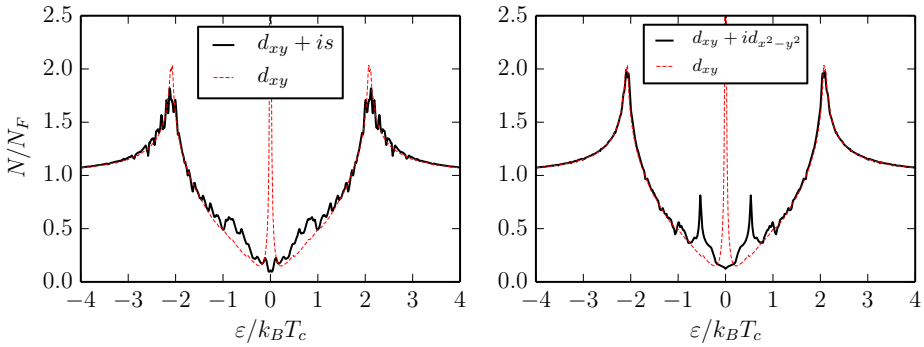

Figure 5.7: *DOS spectrum for a superconducting square grain at $T = 0.3T_c$, with a subdominant order parameter ($T_{c,sub} = 0.5T_c$) alongside the dominant pairing mechanism, i.e. $d_{xy}+is$ pairing. The subdominant order parameter emerges primarily in the regions where the dominant order parameter is suppressed, here along the edges but also in vortex cores, if present. The possibility of Andreev bound states formation is now greatly diminished, and the midgap states are pushed away from the zero (Fermi) energy.*

the dominant order parameter is suppressed somewhere inside grain, due to e.g. pair-breaking or vortices, there is room for a subdominant order parameter to emerge. In Figure 5.9 we see the result of a simulation of a d+is system, where the subdominant s-wave symmetry has appeared where the d-wave order parameter is weakened.

5.6 Spontaneous time-reversal symmetry breaking

The formation of a subdominant order parameter is not the only mechanism in which the midgap states can be shifted. In [44, 46–48] it has been shown that a spontaneous current can emerge along the edges when $T \sim (\xi_0/\lambda_0)\, T_c$, breaking time-reversal symmetry. The current induces a magnetic field which in turn generates screening supercurrents counterflowing inside the bulk. The generated superfluid momentum causes the midgap states to be doppler shifted by $\boldsymbol{v}_F \cdot \boldsymbol{p}_s$, thus lowering the free energy. Many experiments support the existence of a spontaneous low-temperature phase [14–19]. The currents mentioned in [47] were thought to be circulating along the edges. However, finding clear evidence on net circulating currents have proved to be difficult so far.

Setting up such a system (d-wave with pair-breaking edges) with our numerical tool indeed revealed a time-reversal symmetry breaking and transition into a current carrying state at a modestly low temperature of $T \approx 0.18T_c$. This is

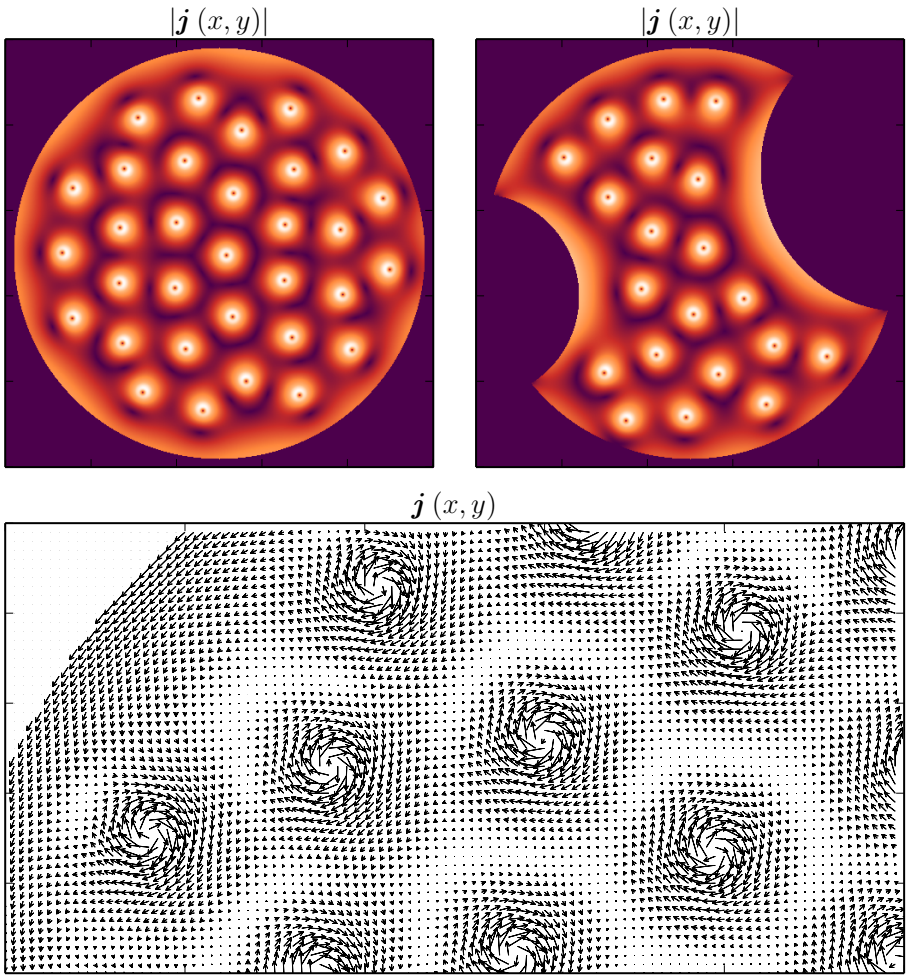

Figure 5.8: *Illustration of the Abrikosov vortex lattice for two different geometries* $\sim 20\xi_0$ *across. The superconductors have s-wave pairing with an applied external magnetic field in the z-direction (into the paper). The top two images are intensity plots of the magnitude of the current. The bottom quiver plot is a crop from the top left system, showing the current direction. Along the boundary of the disc is a current flowing in the opposite direction to that of the vortices. This is the diamagnetic response stemming from the part of the external magnetic field which has not penetrated the interior as flux vortices. Similar systems have been studied by Peeters et al. [45].*

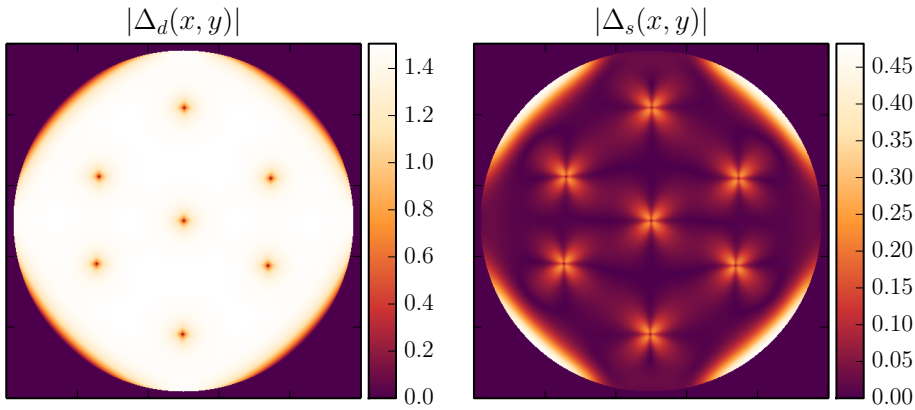

Figure 5.9: *Order parameter components of a system with $d_{x^2-y^2}+is$ pairing symmetry, harboring 7 flux vortices. The left plot shows the d-component, and to right the subdominant s-component. Here $T = 0.2T_{c,d}$, and the critical temperature of the s-component is $T_{c,s} = 0.2T_{c,d}$.*

high in comparison to the previously predicted one, considering that $\xi_0/\lambda_0 \approx 0.01$ for YBCO. Moreover, the currents manifested as "packets" of locally circulating current, where each packet circulates in opposite direction to that of its neighbors (Figure 5.11a). These circulating currents form something resembling a necklace of vortices along any pair-breaking edges (Figure 5.10). It should be mentioned that these are not related to Abrikosov vortices, as they carry a significantly lower magnetic flux at $\pm 10^{-5}\Phi_0$ (Figure 5.11b), and they do not destroy the superconductivity in the center. Moreover, this process also gains some condensate energy near the edges. Plotting $|\Delta(x)|$ for any fixed y-coordinate, we see in Figure 5.11d that the magnitude of the order parameter is always greater in the current carrying state for any x.

This result, if correct, provides a possible explanation as to why these spontaneously generated surface current have been absent in measurements. The resolution limit of current measurement tools are typically much bigger than the size of our circulating currents. If one tries to measure the net current over a large area of mutually countercirculating currents, the net result will naturally be zero. To verify the existence of the necklace currents, a measurement tool with much higher spatial resolution is required. Recently, however, a superconducting quantum interference device (SQUID) with a loop diameter of 46 nm have been constructed [49]. This device, or future improvements of it, might be sensitive enough to find evidence of the edge current phase.

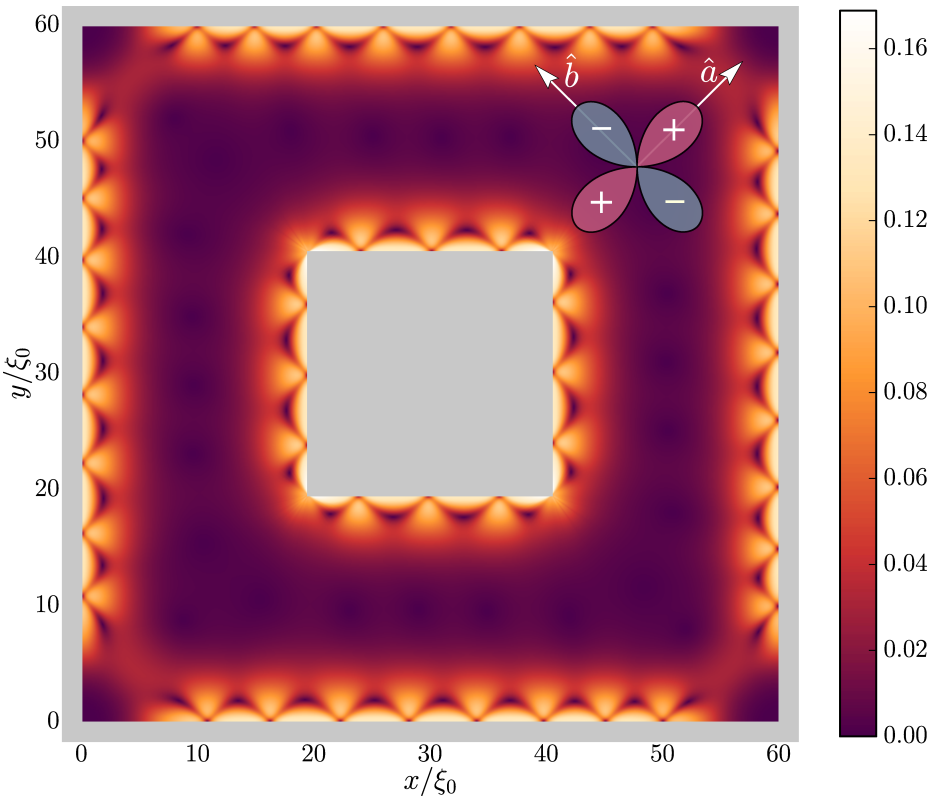

Figure 5.10: *The system investigated in paper I; a d-wave superconducting grain at* $T = 0.1T_c$. *The intensity plot maps the magnitude of the current, in terms of the depairing current, and the symbol in the top right corner illustrates the atomic lattice alignment. The currents appear everywhere the order parameter is suppressed due to pair-breaking, when* $T < 0.18T_c$. *Note, that here* $\xi_0 = \hbar v_F/(2\pi k_B T_c)$.

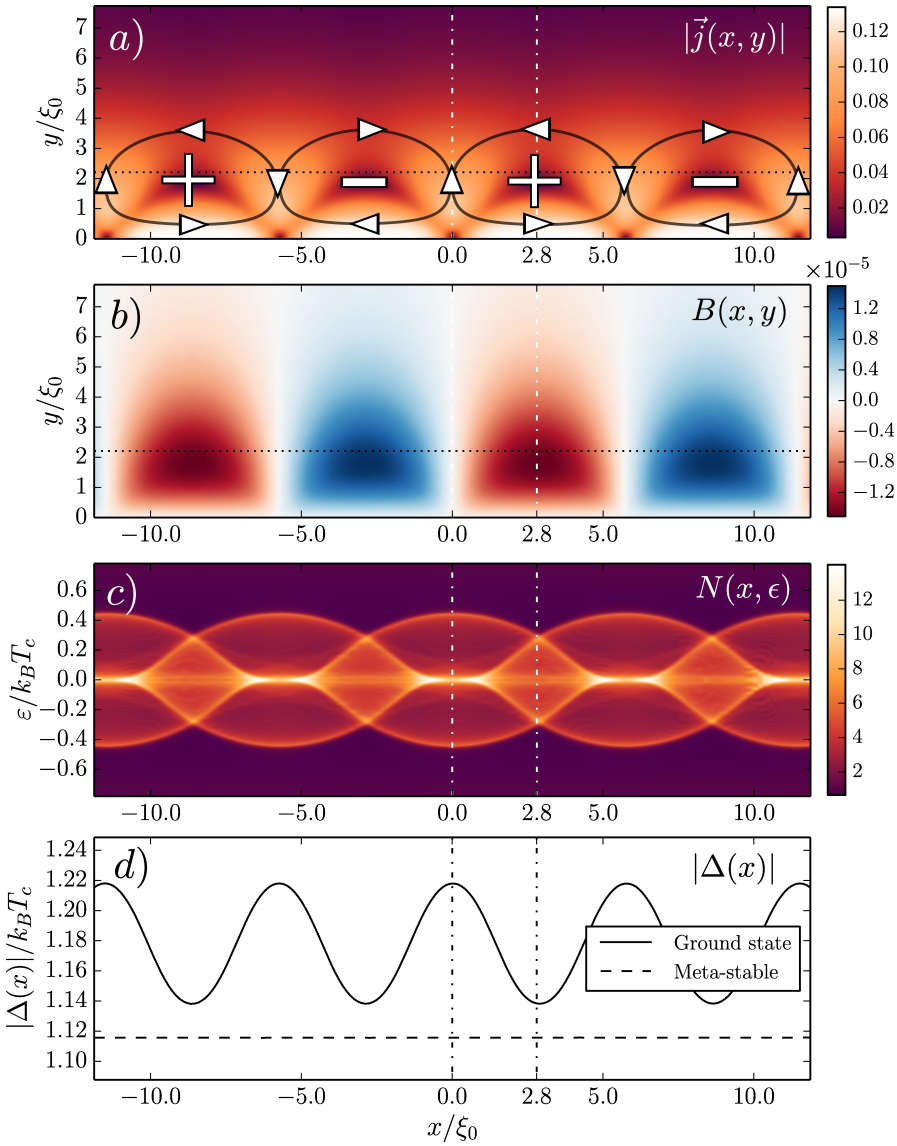

Figure 5.11: *a) Intensity plot of $|\boldsymbol{j}(x,y)|$ along a pair-breaking edge in a d-wave superconductor. Along the edges circulating and counter circulating currents form. These edge currents induce a doppler-shift of the midgap states, pushing them away from the zero energy level, thus lowering the free energy. b) The induced magnetic field. c) LDOS along the edge (as indicated by the dotted line in a)) of the grain. d) One can find a meta-stable state where the edge currents have yet to be formed. Comparing the order parameter to that of the ground state (with edge currents), we see that the induced edge currents have the effect of lifting some of the order parameter suppression which occurs along the edges. Note, that here $\xi_0 = \hbar v_F/(2\pi k_B T_c)$.*

Chapter 6

Summary and Outlook

In this thesis we have developed the basis for a sophisticated software package, or application programming interface (API), to compute solutions to the Eilenberger equation for a finite size thin film (2d) superconducting grain. Similar code libraries exist for other theories of superconductivity, e.g. Ginzburg-Landau theory, but to the best of our knowledge, not for quasiclassical theory. Furthermore, our implementation allows for a completely general boundary definition of the grain, a feature we believe no one has previously published results on. To co-exist with the simulation software, we have also written a number of tools to facilitate management and analysis of the data generated, most notably a visualization tool which allows the user to interactively scan and probe large quantities of data. Moreover, the authored API has been used extensively to investigate many different superconducting systems, and to generate a large body of data for these in order to get a comprehensive picture of the physics behind the results. One particularly interesting result was the emergence of a spontaneous current carrying low temperature phase in a d-wave superconductor, which is the base for paper I.

Suggested uses of the API are to use on its own to study properties of arbitrary superconducting systems, or as a complementary tool to aid in understanding experimental data.

The API consists of a number of different separated blocks of code, most often grouped into C++ classes. From these classes, the user creates (instantiates) a number of objects, representing different numerical and physical qualities, which together to form the physical system to compute for. The naming and modularity of the API is designed to closely resemble that of the physical language. This approach was taken because it allows for code reuse, greatly reduces initial effort for setting up numerical computations, extensibility of software, and it provides a verified code to use, eliminating bug tracking. And since the user does not have to delve into large amounts of low level code, the hope is that this API will open

up possibilities for any researcher willing to learn some rudimentary C++.

While the modular front end is written in C++, the computational backbone is written in CUDA C in order to utilize the great amount of parallelism inherent in the numerical formulation of the problem. However, one does not have to know CUDA to use the API, since this part is fixed and hidden from the user.

The initial development of the API required a significant time investment, as it is hard to envision a design from start which satisfies all demands. Also challenging is to not only design the API to work efficiently with the features planned, but also to predict and allow for future unknown additions. But when in a working reliable form, setting up new systems to compute for only takes a fraction of the time compared to coding from scratch, and requires far less programming knowledge. Another goal with the API was that changes should be relatively easy to make, and mainly affect code in the same class.

In paper I, we investigate a d-wave superconductor in a low temperature regime. Below a certain temperature, the superconductor enters a phase where circulating currents appear spontaneously along any edges where pair-breaking, and thus midgap states, occur. Looking at the free energy, we deduce the transition is of second order. As a result of the emergence of currents and superflow, the unfavorable midgap states are offset from the Fermi energy by a doppler-like shift. The existence of such a low-temperature phase has been predicted by [47], but there are two anomalies to the phase discovered in this thesis. First, the transition temperature is much higher at $T \approx 0.18T_c$ than the predicted one at $\sim 0.01T_c$. Second, there is not a net circulating current, but areas of circulating and counter-circulating currents, yielding a zero net flow. The mechanism behind this peculiar behavior is still largely unknown to us, and we do not know what determines the characteristic and constant size of these "vortices". However, the appearance of spontaneous currents in the form of edge vortices with zero net flow could explain why the predicted low-temperature current carrying phase have not been observed in measurements; any measurement with much coarser spatial resolution than our currents would net to approximately zero. Thus, an instrument with exceptionally fine resolution is required to verify the possible existence of a spontaneous current carrying phase, if manifested as we predict with our calculations.

Looking to the future in terms of code developing, there is much interesting physics to add to the existing work. To begin with, only singlet pairing is implemented, where the coherence functions can be represented by a complex scalar. For p-wave symmetry, requiring triplet pairing, the coherence functions need to have a full 2x2 spin matrix representation, and, naturally, a corresponding solver class derived from `RiccatiSolver`.

Currently, only specular boundary condition is available at the computational domain border. Additionally, while not a boundary condition in the mathematical sense, internal grain boundaries can currently be modeled by defining a spatially varying atomic lattice orientation. This allows for π-junctions like in a Josephson

junction. Still, it would open up possibilities to simulate a much wider range of superconducting systems if mixed type boundary conditions was implemented. Parts of the grain could then be modeled to be in contact with a reservoir or an attached lead with a given fixed superconducting phase.

Other features to implement are time dependent solutions, impurities, and x-functions (particle distribution functions for calculating e.g. non-equilibrium systems).

Turning to the computational aspects, there are a few interesting additions we can think of also. One is to try a different numerical approach to solve the coherence functions, namely by a suitably chosen Runga-Kutta method. These methods are widely used and well documented recipes to solve ordinary differential equations, especially the 4th order variant which provides high numerical accuracy. Another very useful feature to implement would be multi-GPU functionality. Considering the degree of computational parallelism in the problem, several GPUs can share the workload of one task with reasonably good scaling. For minimum data exchange between the GPUs, we suggest the task is split over energy levels/Matsubara frequencies. With this approach each GPU creates a partial solution of e.g. the order parameter for its given set of energy values. The complete solution is then constructed by simply adding the partial solutions.

In conclusion, the developed API allows for quick modeling and simulation of complex superconducting systems, and with the possibility to define a finite size grain with a completely general boundary shape, new ground have been broken. While interesting phenomena can be found with the current feature set of the code, a few carefully chosen additions will likely expose a rich landscape of new physics to discover.

Appendix A

Solutions to the Riccati equations

Riccati formalism states that an equation written on the form

$$\partial_x \gamma = q_0(x) + q_1(x)\gamma + q_2(x)\gamma^2 \tag{A.1}$$

has the following solutions

$$\gamma = \gamma_h + \frac{1}{z(x)} \tag{A.2}$$

where γ_h is known (the particular solution), and $z(x)$ is a solution to

$$z' + [q_1(x) + 2q_2(x)\gamma_h]z = -q_2(x) \tag{A.3}$$

To find $z(x)$, we rearrange (2.14) to the form of (A.1), which gives

$$\begin{array}{rcl} \partial_x \gamma & = & +i\Delta + 2i\epsilon\gamma + i\Delta^*\gamma^2 \\ \partial_x \tilde{\gamma} & = & -i\Delta^* + 2i\epsilon\tilde{\gamma} - i\Delta\tilde{\gamma}^2 \end{array} \tag{A.4}$$

Solving for γ

Identifying the q-terms defined in (A.1) for the equation for γ in (A.4) we find

$$\begin{array}{rcl} q_0 & = & i\Delta \\ q_1 & = & 2i\epsilon \\ q_2 & = & i\Delta^* \end{array} \tag{A.5}$$

Inserting these in (A.3) gives

$$z' + [2i\epsilon + 2i\Delta^*\gamma_h]z = -i\Delta^* \tag{A.6}$$

For the particular solution γ_h, the bulk solution is used

$$\gamma_h = -\frac{\Delta}{\epsilon + i\Omega}, \quad \Omega \equiv \sqrt{|\Delta|^2 - \epsilon^2} \tag{A.7}$$

resulting in the differential equation (via some algebra)

$$\begin{aligned} z' + [2i\epsilon + 2i\Delta^*(-\frac{\Delta}{\epsilon + i\Omega})]z &= -i\Delta^* \\ z' - 2\Omega z &= -i\Delta^* \end{aligned} \tag{A.8}$$

For constant Δ in x, a solution to (A.8) is

$$z(x) = -\frac{\Delta^*}{2i\Omega} + \frac{1}{2i\Omega C}e^{2\Omega x} \tag{A.9}$$

where C is a constant yet to be defined. Verification of solution, using (A.9) in (A.8):

$$\left[\text{with } \; z' = \frac{1}{iC}e^{2\Omega x}\right]$$

$$\frac{1}{iC}e^{2\Omega x} - 2\Omega\left(-\frac{\Delta^*}{2i\Omega} + \frac{1}{2i\Omega C}e^{2\Omega x}\right) = -i\Delta^* \tag{A.10}$$

$$\frac{\Delta^*}{i} = -i\Delta^* \quad \text{OK.}$$

To determine C, boundary condition at $x = 0$ is used

$$\begin{aligned} \gamma(0) &= \gamma_h + \frac{1}{z(0)} \\ \gamma(0) - \gamma_h &= \left[\frac{1}{2i\Omega}\left(-\Delta^* + \frac{1}{C}\right)\right]^{-1} \\ 2i\Omega &= (\gamma(0) - \gamma_h)\left[-\Delta^* + \frac{1}{C}\right] \\ \frac{1}{C} &= \frac{2i\Omega}{\gamma(0) - \gamma_h} + \Delta^* \\ \frac{1}{C} &= \frac{2i\Omega + \Delta^*(\gamma(0) - \gamma_h)}{\gamma(0) - \gamma_h} \end{aligned} \tag{A.11}$$

Now the full expression for γ can be written as

$$\gamma(x) = \gamma_h + \frac{2i\Omega C}{e^{2\Omega x} - \Delta^* C} = \gamma_h + \frac{2i\Omega C e^{-2\Omega x}}{1 - \Delta^* C e^{-2\Omega x}} \tag{A.12}$$

Solving for $\tilde{\gamma}$

Analogously with the previous section, identifying the q-terms for the equation with $\tilde{\gamma}$ (A.4) we find

$$\begin{aligned} q_0 &= -i\Delta^* \\ q_1 &= 2i\epsilon \\ q_2 &= -i\Delta \end{aligned} \tag{A.13}$$

Inserting these in (A.3) gives

$$z' + [2i\epsilon + 2i\Delta^*\tilde{\gamma}_h]z = i\Delta \tag{A.14}$$

To find $\tilde{\gamma}_h$, the *tilde symmetry* [30] is used

$$\tilde{\gamma}(\epsilon,\boldsymbol{p}) = [\gamma(-\epsilon^*, -\boldsymbol{p})]^*, \tag{A.15}$$

and in this case with $\epsilon \equiv i\varepsilon_n - \boldsymbol{v}_F \cdot \boldsymbol{A}$. The sign change originates from the fact that $\boldsymbol{p} \to -\boldsymbol{p}$ for $\tilde{\gamma}$. Inserting this into the homogeneous solution, we get

$$\begin{aligned} \tilde{\gamma}_h &= -\left[\frac{\Delta}{-(i\varepsilon_n)^* - \boldsymbol{v}_F \cdot \boldsymbol{A} + i\sqrt{|\Delta|^2 - [-(i\varepsilon_n)^* - \boldsymbol{v}_F \cdot \boldsymbol{A}]^2}}\right]^* = \\ &= \frac{\Delta^*}{i\varepsilon_n + \boldsymbol{v}_F \cdot \boldsymbol{A} + i\sqrt{|\Delta|^2 - [i\varepsilon_n + \boldsymbol{v}_F \cdot \boldsymbol{A}]^2}} \end{aligned} \tag{A.16}$$

From here on, the notation $\epsilon \equiv i\varepsilon_n + \boldsymbol{v}_F \cdot \boldsymbol{A}$ is again used. Continuing as before, simplifying equation (A.14) gives

$$\begin{aligned} z' + [2i\epsilon + 2(-i\Delta)(\frac{\Delta^*}{\epsilon + i\Omega})]z &= i\Delta \\ z' - 2\Omega z &= i\Delta \end{aligned} \tag{A.17}$$

Again, provided Δ is constant, a solution to (A.17) is

$$z(x) = \frac{\Delta}{2i\Omega} + \frac{1}{2i\Omega\tilde{C}}e^{2\Omega x} \tag{A.18}$$

where $\tilde{C}$ is a constant yet to be defined. Verification of solution, using (A.18) in (A.17):

$$\left[\text{with } \; z' = \frac{1}{i\tilde{C}} e^{2\Omega x}\right]$$

$$\frac{1}{i\tilde{C}} e^{2\Omega x} - 2\Omega \left(\frac{\Delta^*}{2i\Omega} + \frac{1}{2i\Omega\tilde{C}} e^{2\Omega x} \right) = i\Delta \tag{A.19}$$

$$-\frac{\Delta}{i} = i\Delta \quad \text{OK.}$$

To determine $\tilde{C}$, boundary condition at $x = 0$ is used

$$\begin{aligned}
\tilde{\gamma}(0) &= \tilde{\gamma}_h + \frac{1}{z(0)} \\
\tilde{\gamma}(0) - \tilde{\gamma}_h &= \left[\frac{1}{2i\Omega} \left(\Delta + \frac{1}{\tilde{C}} \right) \right]^{-1} \\
2i\Omega &= (\tilde{\gamma}(0) - \tilde{\gamma}_h) \left[\Delta + \frac{1}{\tilde{C}} \right] \qquad \text{(A.20)} \\
\frac{1}{\tilde{C}} &= \frac{2i\Omega}{\tilde{\gamma}(0) - \tilde{\gamma}_h} - \Delta \\
\frac{1}{\tilde{C}} &= \frac{2i\Omega - \Delta(\tilde{\gamma}(0) - \tilde{\gamma}_h)}{\tilde{\gamma}(0) - \tilde{\gamma}_h}
\end{aligned}$$

Now the full expression for $\tilde{\gamma}$ can be written as

$$\tilde{\gamma}(x) = \tilde{\gamma}_h + \frac{2i\Omega\tilde{C}}{e^{2\Omega x} + \Delta\tilde{C}} = \tilde{\gamma}_h + \frac{2i\Omega\tilde{C}e^{-2\Omega x}}{1 + \Delta\tilde{C}e^{-2\Omega x}} \tag{A.21}$$

Bibliography

[1] H. Kamerlingh Onnes, "The resistance of pure mercury at helium temperatures", Commun. Phys. Lab. Univ. Leiden **12**, 120 (1911).

[2] M. Tinkham, *Introduction to superconductivity* (McGraw-Hill, 1975).

[3] J. Bardeen, L. N. Cooper, and J. R. Schrieffer, "Theory of superconductivity", Phys. Rev. **108**, 1175 (1957).

[4] J. G. Bednorz and K. A. Müller, "Possible highT_c superconductivity in the Ba-La-Cu-O system", English, Zeitschrift für Physik B Condensed Matter **64**, 189 (1986).

[5] M. K. Wu, J. R. Ashburn, C. J. Torng, P. H. Hor, R. L. Meng, L. Gao, Z. J. Huang, Y. Q. Wang, and C. W. Chu, "Superconductivity at 93 K in a new mixed-phase Y-Ba-Cu-O compound system at ambient pressure", Phys. Rev. Lett. **58**, 908 (1987).

[6] D. A. Wollman, D. J. Van Harlingen, W. C. Lee, D. M. Ginsberg, and A. J. Leggett, "Experimental determination of the superconducting pairing state in YBCO from the phase coherence of YBCO-Pb dc SQUIDs", Phys. Rev. Lett. **71**, 2134 (1993).

[7] C. C. Tsuei, J. R. Kirtley, C. C. Chi, L. S. Yu-Jahnes, A. Gupta, T. Shaw, J. Z. Sun, and M. B. Ketchen, "Pairing symmetry and flux quantization in a tricrystal superconducting ring of $YBa_2Cu_3O_{7-\delta}$", Phys. Rev. Lett. **73**, 593 (1994).

[8] D. J. Scalapino, E. Loh, and J. E. Hirsch, "*d*-wave pairing near a spin-density-wave instability", Phys. Rev. B **34**, 8190 (1986).

[9] A. A. Abrikosov, "On the magnetic properties of superconductors of the second group", Soviet Physics JETP **5**, 1174 (1957).

[10] C. Caroli, P. G. De Gennes, and J. Matricon, "Bound fermion states on a vortex line in a type II superconductor", Physics Letters **9**, 307 (1964).

[11] J. A. Sauls and M. Eschrig, "Vortices in chiral, spin-triplet superconductors and superfluids", New Journal of Physics **11**, 075008 (2009).

[12] M. Fogelström, "Structure of the core of magnetic vortices in d-wave superconductors with a subdominant triplet pairing mechanism", Phys. Rev. B **84**, 064530 (2011).

[13] P. G. De Gennes, *Superconductivity of metals and alloys* (Addison-Wesley Publishing Company, 1966).

[14] M. Covington, M. Aprili, E. Paraoanu, L. H. Greene, F. Xu, J. Zhu, and C. A. Mirkin, "Observation of surface-induced broken time-reversal symmetry in $YBa_2Cu_3O_7$ tunnel junctions", Phys. Rev. Lett. **79**, 277 (1997).

[15] K. Krishana, N. P. Ong, Q. Li, G. D. Gu, and N. Koshizuka, "Plateaus observed in the field profile of thermal conductivity in the superconductor $Bi_2Sr_2CaCu_2O_8$", Science **277**, 83 (1997).

[16] Y. Dagan and G. Deutscher, "Doping and magnetic field dependence of in-plane tunneling into $YBa_2Cu_3O_{7-x}$: possible evidence for the existence of a quantum critical point", Phys. Rev. Lett. **87**, 177004 (2001).

[17] R. S. Gonnelli, A. Calzolari, D. Daghero, L. Natale, G. A. Ummarino, V. A. Stepanov, and M. Ferretti, "Evidence for pseudogap and phase-coherence gap separation by Andreev reflection experiments in $Au/La_{2-x}Sr_xCuO_4$ point-contact junctions", The European Physical Journal B - Condensed Matter and Complex Systems **22**, 411 (2001).

[18] G. Elhalel, R. Beck, G. Leibovitch, and G. Deutscher, "Transition from a mixed to a pure d-wave symmetry in superconducting optimally doped $YBa_2Cu_3O_{7-x}$ thin films under applied fields", Phys. Rev. Lett. **98**, 137002 (2007).

[19] D. Gustafsson, D. Golubev, M. Fogelstrom, T. Claeson, S. Kubatkin, T. Bauch, and F. Lombardi, "Fully gapped superconductivity in a nanometre-size $YBa_2Cu_3O_{y-\delta}$", Nature Nanotech. **8**, 25 (2013).

[20] C. C. Tsuei and J. R. Kirtley, "Pairing symmetry in cuprate superconductors", Rev. Mod. Phys. **72**, 969 (2000).

[21] R. Carmi, E. Polturak, G. Koren, and A. Auerbach, "Spontaneous macroscopic magnetization at the superconducting transition temperature of $YBa_2Cu_3O_{7-\delta}$", Nature **404**, 853 (2000).

[22] W. Neils and D. Van Harlingen, "Experimental Test for Subdominant Superconducting Phases with Complex Order Parameters in Cuprate Grain Boundary Junctions", Physical Review Letters **88** (2002).

[23] J. R. Kirtley, C. C. Tsuei, A. Ariando, C. J. M. Verwijs, S. Harkema, and H. Hilgenkamp, "Angle-resolved phase-sensitive determination of the in-plane symmetry in $YBa_2Cu_3O7 - \delta$", Nature Phys. **2**, 353 (2006).

[24] H. Saadaoui, G. D. Morris, Z. Salman, Q. Song, K. H. Chow, M. D. Hossain, C. D. P. Levy, T. J. Parolin, M. R. Pearson, M. Smadella, D. Wang, L. H. Greene, P. J. Hentges, R. F. Kiefl, and W. A. MacFarlane, "Search for broken time-reversal symmetry near the surface of superconducting $YBa_2Cu_3O_{7-\delta}$ films using β-detected nuclear magnetic resonance", Phys. Rev. B **83**, 054504 (2011).

[25] A. B. Vorontsov, "Broken translational and time-reversal symmetry in unconventional superconducting films", Phys. Rev. Lett. **102**, 177001 (2009).

[26] Y. Nagai, K. Tanaka, and N. Hayashi, "Quasiclassical numerical method for mesoscopic superconductors: Bound states in a circular d-wave island with a single vortex", Phys. Rev. B **86**, 094526 (2012).

[27] G. Eilenberger, "Transformation of Gorkov's equation for type II superconductors into transport-like equations", Zeitschrift für Physik **214** (1968).

[28] A. I. Larkin and Y. N. Ovchinnikov, "Quasiclassical method in the theory of superconductivity", Soviet Physics JETP **28**, 1200 (1969).

[29] J. W. Serene and D. Rainer, "The quasiclassical approach to superfluid 3He", Physics Reports **101**, 221 (1983).

[30] M. Eschrig, "Scattering problem in nonequilibrium quasiclassical theory of metals and superconductors: General boundary conditions and applications", Phys. Rev. B **80**, 134511 (2009).

[31] Y. Nagato, K. Nagai, and J. Hara, "Theory of the Andreev reflection and the density of states in proximity contact normal-superconducting infinite double-layer", English, Journal of Low Temperature Physics **93**, 33 (1993).

[32] N. Schopohl and K. Maki, "Quasiparticle spectrum around a vortex line in a d-wave superconductor", Phys. Rev. B **52**, 490 (1995).

[33] N. Wennerdal, *Parallel computations of vortex core structures in superconductors,* Master thesis at Chalmers University of Technology, 43, 2011.

[34] *Introduction to the standard template library*, `https://www.sgi.com/tech/stl/stl_introduction.html`, Accessed: 2015-01-07.

[35] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware", Computer Graphics Forum **26**, 80 (2007).

[36] `http://wr0.wr.inf.h-brs.de/wr/hardware/hardware.html`, Accessed: 2015-01-07.

[37] *Tesla GPU accelerators for servers*, `http://www.nvidia.com/object/tesla-servers.html`, Accessed: 2015-01-07.

[38] *Thrust::CUDA toolkit documentation*, `http://docs.nvidia.com/cuda/thrust/`, Accessed: 2015-01-07.

[39] L. J. Buchholtz, M Palumbo, D Rainer, and J. A. Sauls, "The effect of surfaces on the tunneling density of states of an anisotropically paired superconductor", Journal of Low Temperature Physics (1995).

[40] L. J. Buchholtz, M Palumbo, D Rainer, and J. A. Sauls, "Thermodynamics of a d-wave superconductor near a surface", Journal of Low Temperature Physics (1995).

[41] C.-R. Hu, "Midgap surface states as a novel signature for d_{xa}^2-x_b^2-wave superconductivity", Phys. Rev. Lett. **72**, 1526 (1994).

[42] M. Matsumoto and H. Shiba, "Coexistence of Different Symmetry Order Parameters near a Surface in d-Wave Superconductors I", Journal of the Physical Society of Japan **64**, 3384 (1995).

[43] M. Matsumoto and H. Shiba, "Coexistence of Different Symmetry Order Parameters near a Surface in d-Wave Superconductors II", Journal of the Physical Society of Japan **64**, 4867 (1995).

[44] M. Fogelström, D. Rainer, and J. A. Sauls, "Tunneling into current-carrying surface states of high-T_c superconductors", Phys. Rev. Lett. **79**, 281 (1997).

[45] L. F. Zhang, L. Covaci, M. V. Milošević, G. R. Berdiyorov, and F. M. Peeters, "Vortex states in nanoscale superconducting squares: The influence of quantum confinement", Physical Review B **88**, 144501 (2013).

[46] S. Higashitani, "Mechanism of paramagnetic Meissner effect in high-temperature superconductors", Journal of the Physical Society of Japan **66**, 2556 (1997).

[47] Y. S. Barash, M. S. Kalenkov, and J. Kurkijärvi, "Low-temperature magnetic penetration depth in $\boldsymbol{d}$-wave superconductors: Zero-energy bound state and impurity effects", Phys. Rev. B **62**, 6665 (2000).

[48] T. Löfwander, V. S. Shumeiko, and G. Wendin, "Time-reversal symmetry breaking at Josephson tunnel junctions of purely d-wave superconductors", Phys. Rev. B **62**, R14653 (2000).

[49] D. Vasyukov, Y. Anahory, L. Embon, and D. Halbertal, "A scanning superconducting quantum interference device with single electron spin sensitivity", Nature Nanotechnology **8**, 639 (2013).